STRUT User's Guide

Jean-Marc Boite, Laurent Couvreur, Geoffrey Wilfart

November 4, 2005

Contents

I *STRUT* in a Nutshell **3**

1 Introduction **4**
- 1.1 What Is ASR ? . . . 4
  - 1.1.1 Overview . . . 5
  - 1.1.2 Front-End . . . 5
  - 1.1.3 Back-End . . . 5
- 1.2 What is *STRUT* not? . . . 6
- 1.3 What is *STRUT* ? . . . 6
  - 1.3.1 Overview . . . 6
  - 1.3.2 Front-End . . . 8
  - 1.3.3 Back-End . . . 8
  - 1.3.4 Application Compiler . . . 12
- 1.4 What Next? . . . 12

2 Training Procedure **14**
- 2.1 *STRUT* components . . . 14
- 2.2 Create a Sample File . . . 14
- 2.3 Compute a Feature File . . . 15
- 2.4 Obtain a Segmentation File . . . 15
- 2.5 Train an acoustic model . . . 17
- 2.6 Realignment Process . . . 17

3 Testing Procedure **20**

4 The Tools **24**
- 4.1 Database Handling . . . 24
  - 4.1.1 General Purpose . . . 24
  - 4.1.2 Samples . . . 24
- 4.2 Training and Testing of Models . . . 25

5 StrutSurfer **26**
- 5.1 The *StrutSurfer* Window . . . 26
- 5.2 Basic Functions . . . 26
  - 5.2.1 The *File* Menu . . . 26
  - 5.2.2 The *Edit* Menu . . . 28
- 5.3 ASR Functions . . . 30
  - 5.3.1 The *Strut* Menu . . . 30

5.4 The *Utils* Menu 30
5.5 Advanced Features 30
5.5.1 Computing Features and Probabilities 30
5.5.2 Performing Alignment 33
5.6 Macros 34

II *STRUT* Reference Guide 35

6 The Strut User Interface 36
6.1 Introduction 36
6.2 Case Dependency and Shell Interaction 36
6.3 Argument Types 36
6.4 Common Parameters 39
6.5 Multiple Specification of an Argument 40
6.6 Command Line Help 40

7 The Files 43
7.1 Strut File Headers 43
7.2 Strut File Types 44
7.3 Samples 44
7.4 Features 44
7.5 Labels 45
7.6 Segmentation 46
7.7 Probabilities 46

8 Training Perl Scripts 47

A Install STRUT 49
A.1 Software Installation 49
A.1.1 Install From Sources 49
A.1.2 Install From CVS 49
A.1.3 Install Binary Package 49
A.2 File System Organization 49
A.2.1 Database Subdirectory Organization 50

B Environment Variables 52

C Examples of HMM topology 53
C.0.2 A Word-Based HMM set 55
C.0.3 A Phoneme-Based HMM set 57

Part I

STRUT in a Nutshell

Chapter 1

Introduction

This document has been designed to provide a complete step by step guide to the *Speech Training and Recognition Unified Tool (STRUT)*. The guide assumes that the user has no previous training with the *STRUT* software, but simply has an installed version of the package. Having some experience about speech recognition is of course a big advantage. This user's guide does not provide any theoretical information. There are good tutorials in the litterature.

The user is assumed to have a good knowledge of Unix, since it is the prefered operating system to run the programs. Users should complete the guide sequentially, and within a short time the will have access to all the background knowledge in order to train and run an automatic speech recognizer.

1.1 What Is ASR ?

In this section, we overview what an automatic speech recognition (ASR) system consists in. Advanced readers who are already familiar with ASR can proceed directly to section 1.3. Beginner readers can find more detailed information in the reference books [1, 2].

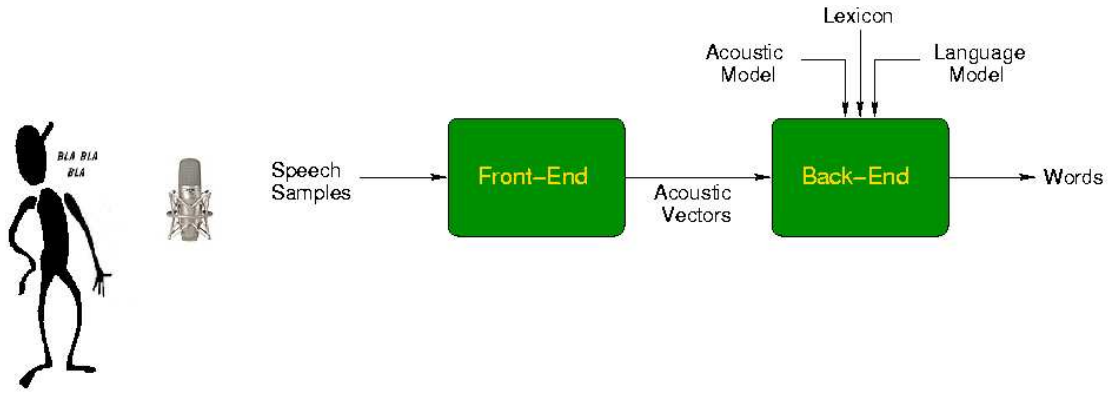


Figure 1.1: A typical ASR system consisting of a microphone, a front-end and a back-end.

1.1.1 Overview

Figure 1.1 outlines the generic structure of an ASR system. For a given speech waveform, the ASR system produces a word string that is the most likely associated with that waveform. The speech waveform is recorded by means of a microphone which converts acoustic pressure into an electrical signal. The speech signal is then sampled. The resulting speech samples are fed into the ASR system which outputs a word string. Typically, an ASR system is divided into a front-end part and a back-end part.

1.1.2 Front-End

The front-end computes relevant information with respect to the ASR process along the speech samples. Practically, a set of coefficients are computed from a frame of speech samples which is typically 20-30 ms long. These coefficients, also called acoustic features, are gathered into the so-called acoustic vector. This vector is intended to represent the spectral contain of the current frame. Next, the frame is shifted by 10-20 ms and the computation is repeated. Eventually, we obtain a sequence of acoustic vectors which represent the temporal evolution of the spectral contain of a given sequence of speech samples.

1.1.3 Back-End

The back-end interprets the sequence of acoustic vectors to find out the pronounced words. This processing is generally called the *decoding* process. Clearly, the problem is very complex since the number of words and the word limits are unknown. The back-end relies on three sources of information to perform its decoding, namely the acoustic model, the lexicon and the language model.

Acoustic Model

Given a sequence of acoustic vectors, the back-end searches to find the sequence of words which has the most likely produced it. To do so, a stochastic model of every possible word in terms of acoustic vectors, the so-called acoustic model, is required. Currently, Hidden Markov Models (HMM) [1] are the most popular acoustic models. According to the HMM formalism, any sequence of acoustic vectors is a piecewise stationary stochastic process for which each stationary segment is associated with a specific state and has statistical properties depending on that state. The sequence of states is controlled by a Markov process.

The acoustic models have to be trained before the recognition takes place. Practically, many sound examples of the words should be presented to the training system. Except when the number of words is limited (*e.g.*, digit recognition), it is practically not possible to train accurately word-based acoustic models. It would require too large speech databases. It is generally preferred to represent speech with speech units smaller than words. Commonly used speech units are phonemes. In most languages, any word can be represented by a sequence of phonemes drawn from a set of about 30–40 phonemes. Hence, it is sufficient to train phoneme-based acoustic models. The word-based acoustic models are then obtained by concatenating the phoneme-based acoustic models according to the phonetic transcriptions contained in a lexicon.

Lexicon

When words are used as speech units, the lexicon is just equivalent to the list of words that can be recognized. When speech units are not words, the lexicon maps the words to sequences of speech units. It indicates how the acoustic models of the speech units should be linked to form the acoustic models of the words.

Language Model

As we mentioned earlier, the most likely sequence of words is searched during the decoding process. In order to limit the number of hypotheses, a language model is used. It contains information about the allowable word sequences.

1.2 What is *STRUT* not?

This is not a dictation system.

1.3 What is *STRUT* ?

In this section, we present how *STRUT* performs the tasks involved in an ASR system. Readers who are already familiar with the components of *STRUT* can proceed directly to chapter 2.

STRUT is a research tool, designed for speech and speaker recognition. While Gaussian Mixture Models (GMM) can be trained, hybrid Multi-layer Perceptron / Hiden Markov Models (HMM/MLP) have been particularily developed.

1.3.1 Overview

STRUT consists in a set of stand-alone programs which can be either run as command lines or included in scripts in order to perform the different tasks required in an ASR system. *STRUT* has been developed as a research tool. It aims at being user-friendly, providing comprehensive help and allowing to access all intermediate results of the ASR process.

When used in a normal mode, the components of *STRUT* exchange information via files with a particular format. Here is the structure of such a file (a sample file in this case):

```
STRUT_2A
  1152
file_type -s7 samples
database_id -s6 enu
database_version -s3 aurora2
strut_release -s8 strut2.0
strut_version -s8 internal
data_format -s12 LittleEndian
data_offset -i 1204
data_size -i 49867
machine -s4 i686
nodename -s17 mutlu.multitel.be
os_release -s11 2.4.8-26mdk
os_version -s32 #1 Sun Sep 23 17:06:39 CEST 2001
```

```
    sample_coding -s7 shorten
    sample_n_bytes -i 1
    sample_rate -i 8000
    step_count -i 2
    sysname -s5 Linux
    utterance_count -i 3
    utterance_start_offset -i 1188
    step_1_command_line -s153 data-format=big-endian data-size=2
          database-id=enu database-version=aurora2
          file-type=samples(sample-rate=8000)
          input=fkk_849z339a.08 output=fkk_849z339a
    step_1_program_date -s44 internal - executed Tue Dec  3 14:27:32 2002
    step_1_program_name -s8 strutify
    step_2_command_line -s45 dir=aurora2 format=samples output=aurora2.sam
    step_2_program_date -s44 internal - executed Tue Dec  3 14:30:05 2002
    step_2_program_name -s14 create-archive
    end_head
    5678901234567890123456789012345678901234567890123456789012345678901234
    567890123456789012345678901234567890123456789012345678901234567890123456789012345678901


    mlt_9242a
    mfc_7521a
    fkk_849z339a

... # utterance data offset #


... # block data #
```

We present in section 2 how to obtain such a file. It begins with an ASCII header. This header starts with the `STRUT_2A` tag which identifies the file as a *STRUT* file, then it gives the header size in bytes and a set of fields which characterize the information contained in the file. For instance, the fields can define the type of file (`file_type`), the sampling frequency in Hz (`sample_rate`), the data byte ordering (`data_format`), the file history (`step_*` lines) or the number of utterances (`utterance_count`). Indeed, *STRUT* supports archive files which can contain data for several speech utterances. In this case, every utterance is uniquely identified by an utterance identifier. The list of utterance identifiers is given after the header. Then, the file contains the utterance data offsets: a sequence of binary coded integers which locate the beginning of every utterance in the data block of the file. Finally, we find the sample data which are stored sequentially for all the utterances.

In the following, we describe how to process that file to actually perform the ASR tasks. More details can be found in section 2.

1.3.2 Front-End

There exists several approaches to extract acoustic features from the speech samples. Most popular front-ends compute cepstral-like coefficients for every analysis frame. More especially, two sets of coefficients are often used, namely Mel-frequency cepstral coefficients (MEL) [3] and Perceptual Linear Predictive coefficients (PLP) [4]. Those coefficients can be computed by means of the `extract-features` program. The processing is also integrated in programs like `mlp-train` and `recognize`, so you don't need to pre-compute the features

First, every speech utterance from the sample file is sliced into frames which are pre-emphasized by high-pass filtering and multiplied by a Hamming window. Then, a Fourier analysis is performed over every sample frame and the power spectrum is computed. Next, an auditory spectrum is obtained by applying a non-uniform filterbank. Finally, cepstral coefficients are derived from the auditory spectrum.

Additional processing is possible to obtain acoustic features which are less sensitive to additive and convolutional noises during operation. So far, it includes logRASTA and jahRASTA processings [5] as well as Cepstral Mean Subtraction (CMS) [6], Spectral Subtraction (SS) [7] and Wiener filtering [8].

1.3.3 Back-End

The back-end processing in *STRUT* is implemented in the program `recognize`. Actually, it is able to realize the whole ASR process, even computing the front-end. However, you can also compute the frond-end separately, *i.e.*, by applying `mel-cepstrum` or `rasta-plp` to a sample file, and to feed `recognize` with the resulting feature file. Then, `recognize` applies the acoustic model to the acoustic vectors to obtain probability vectors and performs the decoding under the constraints of the lexicon and the language model. The decoding is based on the Viterbi algorithm with pruning techniques [2]. We describe in the following what exactly the acoustic model, the lexicon and the language model consist in.

Acoustic Model

As mentioned earlier, we assume that any sequence of acoustic vectors has been emitted by a certain sequence of HMMs. In order to find the most likely word string, the decoding process searches the most likely sequence of HMMs given the sequence of acoustic vectors and outputs the corresponding word string. The HMMs have to be trained beforehand.

First, we define the topologies of the HMMs, *i.e.*, the number of states and the transition probabilities between the states. Though these parameters can be automatically inferred, we commonly use the left-to-right topology. Figure 1.2(a) depicts the topology of the HMM for the word "`eight`". The left-to-right structure models the sequential mechanism of speech production. The figures on the arcs represent the transition probabilities for leaving a state to another. They sum to one. The state labels indicate which statistical distribution should be considered. The total number of states represents the minimum duration of any sequence of acoustic vectors emitted by that HMM. The self-transitions model the stationary segments. To build a complete ASR application, we need several word HMMs. Their topologies are described in a HMM topology file. Appendix C.0.2 gives the HMM topology file of word-based HMMs for English digit recognition. It is sometimes preferred to use phoneme-based HMMs: any word HMM is obtained by concatenating several phoneme HMMs. Figure 1.2(b) shows the topology of the HMM for the

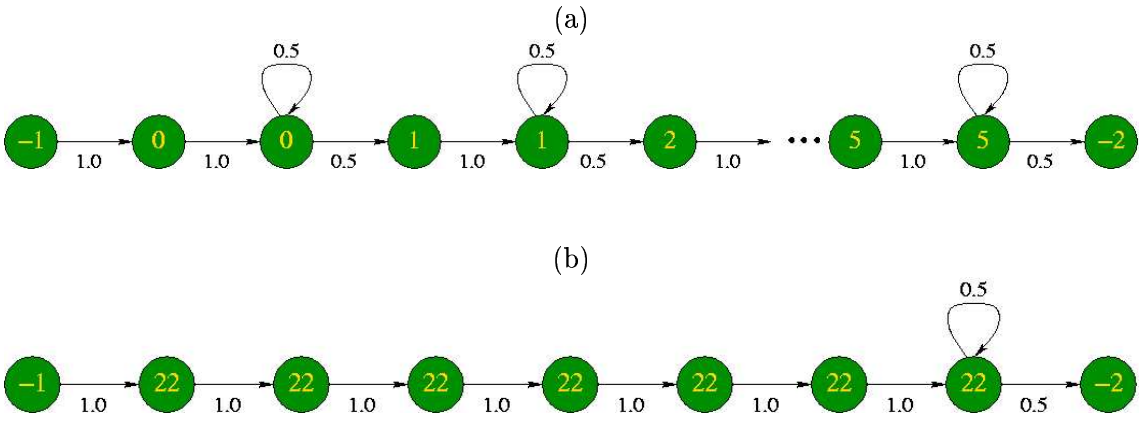


Figure 1.2: Topology of a left-to-right HMM for (a) word "`eight`" and (b) phoneme "`ey`".

phoneme "`ey`", the first phoneme of the word "`eight`". Appendix C.0.3 describes the entire set of English phoneme-based HMMs.

Next, we have to estimate the statistical distributions of every state. Many approaches have been proposed to estimate the state statistical distributions [1, 2]. In *STRUT*, we have adopted the hybrid Hidden Markov Model / Artificial Neural Network (HMM/ANN) paradigm [9]. More especially, we consider a special class of ANNs, namely Multi Layer Perceptrons (MLP). Once they have been properly trained, such statistical tools allow to estimate the *a posteriori* state probability for any acoustic vector. When the recognition takes place, a sequence of acoustic vectors is first computed along the speech utterance to be recognized. Next, the MLP estimates the *a posteriori* state probabilities, all the states at the same time, resulting in a sequence of probability vectors. Finally, the decoding process searches the resulting lattice of probabilities for the most likely path, *i.e.*, with the highest probability, under the constraints defined by the topologies of the HMMs, the lexicon and the language model.

Such a MLP can be efficiently trained in a supervised mode. Given a speech database, we compute the acoustic vectors and we present the MLP with the acoustic vectors and the suited states (equivalently, an ideal probability vector with entries equal to one for the suited state and zero for the other). The weights are then updated in order to minimize the error between the actual outputs and the suited ones. In *STRUT*, the training procedure is performed by running a Perl [10] script which can be generated via a user-friendly interface written in PerlTk [11], `train.ptk`. Note that an acoustic model is always developed for a given language and for a given type of acoustic features (sampling rate, frame shift/length, etc).

Remark. The ASR Chicken and Egg Problem. The performance of an ASR system will depend on the quality of the acoustic model. In order to train accurately an acoustic model, a valid state-by-state segmentation of the training sequence of acoustic vectors is required. Such a segmentation can be generated manually but this work is labor-intensive and tedious since the training speech database can be several hour long and sometimes the states do not mean anything (*e.g.*, for word-based HMMs). Alternatively, one can generate the segmentation automatically but it requires an existing acoustic model. We generally resort to an iterative procedure: the MLP weights and the segmentation are recomputed alternatively until convergence (see figure 1.3). The procedure is called "embedded training". The training of the weights, which is itself an iterative process, is embedded within several re-alignment steps.

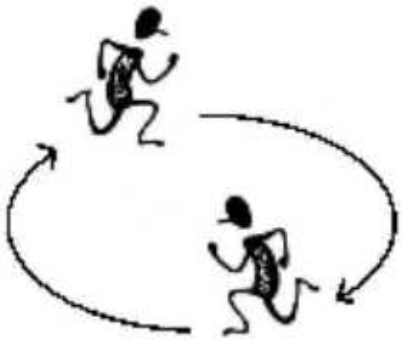


Figure 1.3: The ASR run.

There exists many ways to initialize the procedure. See section 2.6 for more information.

Lexicon

As we mentioned earlier, the lexicon consists in a list of the transcriptions of words in terms of speech units. For example, figure 1.4(a) gives a lexicon for recognition of English digits where the speech units are actually words, *i.e.*, digits. Likewise, figure 1.4(b) gives the lexicon for the same words where the speech units are phonemes. The format of lexicon files is simple: each line contains a word followed by a sequence of symbols corresponding to speech units. Clearly, these speech units should exist in the HMM topology file.

During the decoding, it is possible to consider several different transcriptions for a given word. It helps to handle variations with respect to the canonical transcription like accents, co-articulation effects or mis-pronunciations. There exists syntax rules to concisely generate alternatives from a reference transcription by means of brackets. Besides, it is sometimes also possible to generate the transcriptions automatically. See "Reference Guide of Application Compiler" for a complete description about lexicon syntax rules and the use of automatic phonetization.

Language Model

In the framework of *STRUT*, the language model takes the form of a finite-state grammar (FSG). This grammar defines in a concise hierarchical way all the possible sequence of words. Figure 1.5 shows a FSG file for recognition of English digit sequences. The file begins with the tag `#!FSG` which identifies it as a FSG file. The main macro `<START>` defines the structure of any sentence allowed by the grammar. The definition relies on a set of nested rules. They indicate that the grammar allows sentences containing any number of times an element of type `<DIGIT>`. The macro `<DIGIT>` is defined afterwards. It could contain any of the ten digits. An alternative unit can be recognized instead of a digit, namely the garbage `*`. This particular unit can be seen as a dummy word which models anything except a digit. It is very useful to handle out-of-lexicon words which will be displayed as `UNK` during recognition. See "Reference Guide of Application Compiler" for a complete description about the generation of FSG files.

```
#!Transcriptions

oh oh
zero zero
one one
two two
three three
four four
five five
six six
seven seven
eight eight
nine nine


#!Transcriptions

oh ow
zero z ih r ow
one w ah n
two tcl t uw
three th r iy
four f ao r
five f ay v
six s ih kcl k s
seven s eh v ah n
eight ey tcl t
nine n ay n
```

Figure 1.4: (a) Word-based lexicon (file `aurora2-words.dic`) and (b) phoneme-based lexicon (file `aurora2-phonemes.dic`) for English digit recognition.

1.3.4 Application Compiler

The HMM topology file, the lexicon and the FSG file are human-readable files which define entirely all the sentence models with respect to the decoding process.

In order to prepare the decoding process, we compile the HMM topology file, the lexicon and the FSG file into an application file. This file is obtained by using the program `compile-asr`. Note that the lexicon can be embedded in the FSG file. For example, the transcriptions given either in figure 1.4(a) or in figure 1.4(b) can be simply cut and pasted at the end of the FSG file. See "Reference Guide of Application Compiler" for a complete description about the generation of application files.

1.4 What Next?

If you have never used *STRUT*, you should now read the chapter 6, which explain the general syntax of the *STRUT* programs. Otherwise you can go on the next chapter, a tutorial of the training procedure.

```
#!FSG

<START>=seq(
             rep(
                  alt(
                       <DIGIT>
                       *[UNK]
                  )alt
             )rep
        )seq;

<DIGIT>=alt(
             zero
             oh
             two
             three
             four
             five
             six
             seven
             eight
             nine
        )alt;
```

Figure 1.5: Finite-State Grammar for English digit recognition (file `aurora2.fsg`).

Chapter 2

Training Procedure

2.1 *STRUT* components

The *STRUT* programs allow to create and to manipulate *STRUT* files. Along the process of training and testing an ASR system in the *STRUT* framework, it may sometimes be necessary to check that files are not corrupted: `display-strut` can be used as a validation tool. This program displays the contain of a *STRUT* file utterance by utterance and some information about every utterance. If a *STRUT* file cannot be displayed, it is likely to be corrupted, probably because of a bug in the program that created the file.

If you want to visualize your data, you can also use StrutSurfer. This graphical interface is based on TclTk [12] and Python [13]. These scripting languages should be installed. StrutSurfer allows to navigate seamlessly in a *STRUT* archive file, to visualize and process samples, features or segmentation related to any utterance. See chapter 5 for more information.

A detailed list of the *STRUT* programs can be found in chapter 4

2.2 Create a Sample File

The initial step of the training procedure consists in creating the speech sample archive file. Let's assume that we first copy the 8440 speech data files `speech/data/train/clean/*.08` from cdrom 2/4 of the Aurora2 collection. These files are English digit sequences sampled at 8000 Hz and pronounced by 110 male and female speakers.

Next, we transform these RAW speech files into *STRUT* sample files by adding a header with the command `strutify`. For example,

```
> strutify input=samples/train/fac_13a.08
           output=samples/train/fac_13a data-format=big-endian
           data-size=2 file-type=samples'(sample-rate=8000)'
           database-id=enu database-version=aurora2
```

The `input` and `output` define the RAW speech file and the *STRUT* sample file, respectively. In our example, every speech sample is binary coded on two bytes (`data-size=2`) with the left-most byte being most significant (`data-format=big-endian`). This information allows `strutify` to interpret properly the byte stream and prevent wrong byte ordering. Next, we specify the type of *STRUT* files to create, namely sample file (`file-type=samples`) sampled at 8000 Hz

('(sample-rate=8000)'). Finally, we provide two informative parameters which identify the language (database-id) and the database (database-version).

After removing all the *.08 files, we create the archive sample file train.sam by merging all the files of type samples located in the directory samples/train:

```
> create-archive output=samples/train.sam format=samples
                 dir=samples/train
```

Sometimes, sample files can be quite big. We may want to use the lossless *shorten* compression algorithm. A clean signal can be compressed by a factor 2.

```
> convert-samples input=samples/train.sam output=samples/train.shn
                 coding=shorten
```

2.3 Compute a Feature File

The second step of the training procedure consists in computing the acoustic vectors. Usually, this step will be embedded in the training program, but we will do it seperately here for didactic reasons.

In our example, we have chosen to compute the PLP features. They can be obtained with the following command:

```
> extract-features input=samples/train.shn output=features/train.plp
               feature-type=plp
```

which computes by default 13 PLP cepstral coefficients with frames of 30 ms, shifted by 10 ms. Many other setups are possibles for computing the acoustic features. For example, logRASTA processing can be activated (rasta=log):

```
> extract-features input=samples/train.shn output=features/train.plp
               feature-type=plp rasta=log
```

2.4 Obtain a Segmentation File

As we explained previously, the main problem for training a MLP is to obtained a first segmentation. In our example, we assume that a segmentation is available. That is, every utterance from the training set corresponds to a sequence of English digits, hence a sequence of HMMs, hence a sequence of states. The segmentation file contains the boundaries of every state for every utterance. The segmentation files are located in the segmentation subdirectory. The segmentations are available for both word-based HMMs (segmentation/train-words.seg) and phoneme-based HMMs (segmentation/train-phonemes.seg). See section 2.6 for more information about obtaining a segmentation.

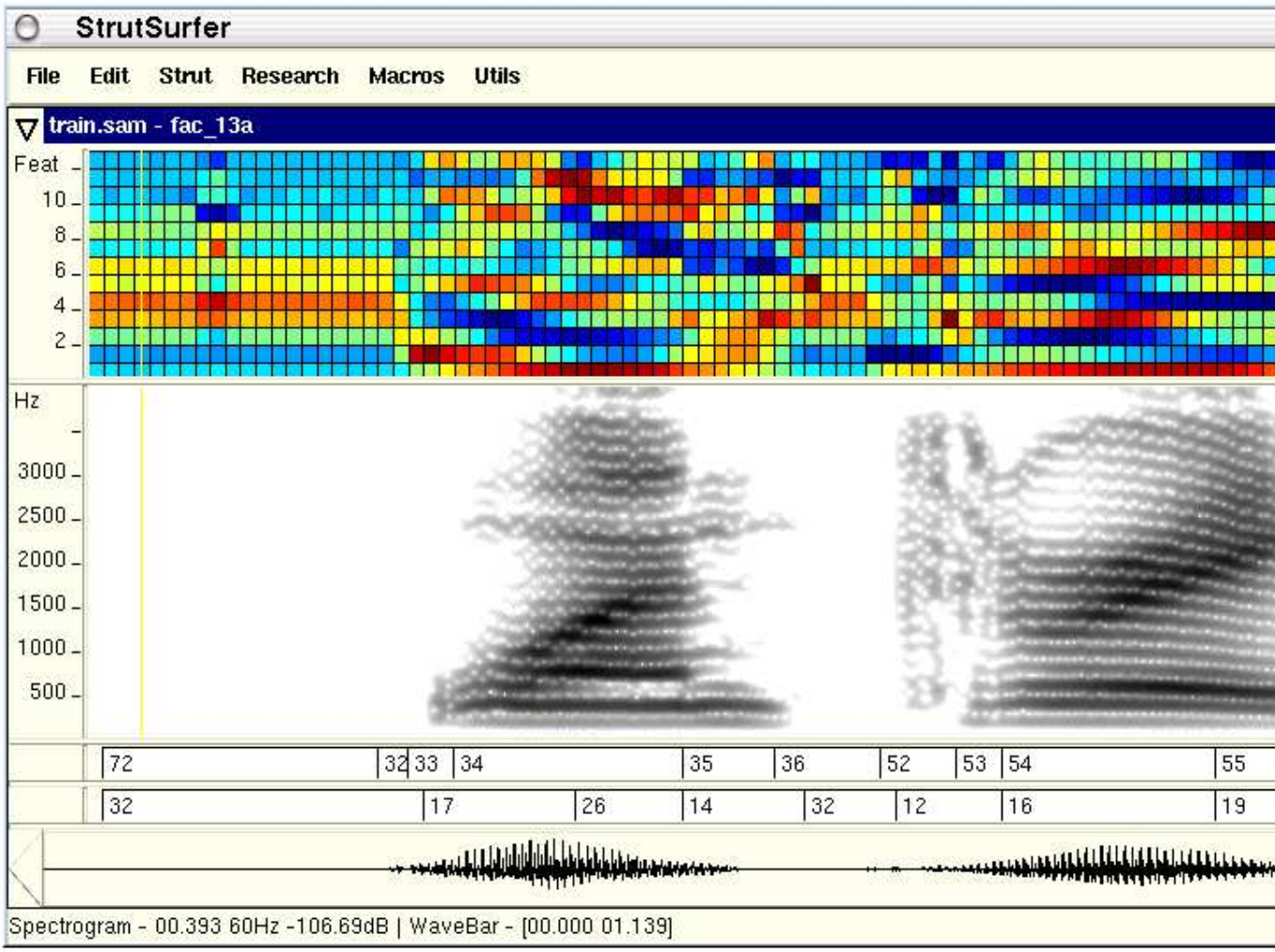


Figure 2.1: Screen-shot of the *StrutSurfer* interface.

2.5 Train an acoustic model

Before training a MLP, it is worth verifying that your segmentation really correspond to your samples. To do so, you can use the graphical interface StrutSurfer (fig 2.1).

First, we open the training sample file (`Select samples/train.sam in the File Open` menu). The file is automatically recognized as a *STRUT* archive sample file. A new window pops up and you are asked to select an utterance, for example `fac_13a`. Then, you can create a new pan and display the spectrogram. Likewise, we can display a color map of the acoustic features. We finally load the corresponding word-based and phoneme-based segmentations. See chapter 5 for more information.

Once we have verified that the sample files and the segmentation files are valid and coherent, we can start training the acoustic models. As we said earlier, the training process consists in estimating a MLP that classifies the acoustic vectors with respect to the states given in the segmentation. It is performed by running a Perl script which can be easily generated with the interface `strut-train.ptk`.

You select the different pages, and fill the fields. Finally, you create the training Perl script (run `Create script`). The interface will tell you if any parameter is missing. If you have filled in all the mandatory parameters, an new window will apppear where you can edit the script. Once you are happy with it, you save it

The script is saved can be executed to generate the MLP.

Figure 2.2 shows the script we use to train a MLP for MEL acoustic features, starting with a models file. Scripts for other training conditions can be generated easily by editing this script. See the YET-TO-BE-WRITTEN"User Guide for `train.ptk`" for a complete description on the MLP training options.

2.6 Realignment Process

As we mentioned in section 1.3.3, the training of an acoustic model requires an state-by-state segmentation. If we assume that such a segmentation is available, it is straighforward to obtain an acoustic model as described above. However, the segmentation may be approximative and the resulting acoustic model is inaccurate. In order to refine the acoustic model, the segmentation should be recomputed: the so-called *realignment* process.

Assume we have a coarse first segmentation, not necessarily in *STRUT* format. First, we need to strutify the segmentation. To do so, we prepare for every utterance a file (for example, located in the `segmentation/train` directory) in which we store in a binary format the following information:

```
<1st state label><1st state right boundary>
|     int2      ||        float4          |

<2nd state label><2nd state right boundary>
|     int2      ||        float4          |

<3rd state label><3rd state right boundary>
|     int2      ||        float4          |
...
```

The segmentation for every utterance is then strutified,

```
#!/usr/bin/perl

push @INC, "/usr/local/asr/strut/lib/perl/modules";

use File::Basename;
require ParseArgs;
require ann::EmbeddedMlpTrain;

$train{alignment_iteration} = "2";
$train{alignment_stop_criterion} = "iteration";
$train{application} = "/asr/database/frf/bref/application/bref120.train.app";
$train{beam_width} = "100";
$train{bunch_size} = "32";
$train{cache_size} = "25000";
$train{divide_by_priors} = "0";
$train{end_with_trailing_silence} = "1";
$train{feature_parameter_source} = "/asr/database/frf/bref/features/bref-16kHz.mel";
$train{input} = "/asr/database/frf/bref/samples/bref120-16kHz.part1.sam";
$train{input_models} = "/asr/database/frf/bref/models/frf16.ff.180-1000-036.plp-lograsta.mlp";
$train{max_word_hypothesis} = "8";
$train{models_dir} = dirname($0);
$train{models_name_pattern} = "frf16.ff.ANN_SIZE.plp-lograsta.mlp";
$train{output_function} = "Sigmoid-X-Entropy";
$train{output_layer_dir} = dirname($0);
$train{percentage_training} = "94";
$train{remove_output_layer} = "1";
$train{shuffle} = "1";
$train{skip_frames_cross} = "2";
$train{skip_frames_train} = "4";
$train{strut_tmp} = "/dev/null";

$train{expert}{nlda}{epochs} = "6";
$train{expert}{nlda}{feature_multipliers}[0] = "1-12";
$train{expert}{nlda}{feature_multipliers}[1] = "0-12";
$train{expert}{nlda}{feature_multipliers}[2] = "0-0";
$train{expert}{nlda}{formula}[0] = "1";
$train{expert}{nlda}{formula}[1] = "-2 -1 0 1 2";
$train{expert}{nlda}{formula}[2] = "2 1 -2 -2 -2 1 2";
$train{expert}{nlda}{frame_selection} = "0 3 6 9 12";
$train{expert}{nlda}{hidden_layer_size} = "500-24";
$train{expert}{nlda}{initial_learning_rate} = "32";
$train{expert}{nlda}{learning_rate_rate} = "50";
$train{expert}{nlda}{output_function} = "Sigmoid-X-Entropy";
$train{expert}{nlda}{window_offset} = "-6";
$train{expert}{nlda}{window_width} = "13";

$train{expert}{top}{epochs} = "6";
$train{expert}{top}{frame_selection} = "0 3 6 9 12 15 18";
$train{expert}{top}{hidden_layer_size} = "1000";
$train{expert}{top}{initial_learning_rate} = "8";
$train{expert}{top}{learning_rate_rate} = "50";
$train{expert}{top}{output_function} = "Sigmoid-X-Entropy";
$train{expert}{top}{window_offset} = "-9";
$train{expert}{top}{window_width} = "19";



$train{train_from} = "models";

#
#  Include options from scripts specified in the command line
#

ARG:
while ($#ARGV >= 0) {
foreach $param ("debug","verbose") {
if ($ARGV[0] eq $param) {$train{$param}=1;shift;next ARG;}
}

do shift @ARGV;
}

if (exists($train{debug}) && ParseArgs::truearg($train{debug})) {
$train{utterance_count} = 100;
$train{utterance_count_xval} = 10;
$train{learning_rate_schedule} = '0.015 0.01 0.008';
}


$mlp = new EmbeddedMlpTrain(\%train);
$mlp -> process;
```

Figure 2.2: Training Perl script.

```
> strutify input=segmentation/train/fac_13a.08
           output=segmentation/train/fac_13a
           add-field='[float(name=analysis_frame_length_ms value=30)
                       float(name=analysis_frame_shift_ms value=10)
                       string(name=label_coding value=segments)
                       int(name=frame_size value=1)
                       int(name=label_count value=1)
                       string(name=label_format value=0)
                       int(name=label_size value=33)
                       string(name=label_type value=Segment)]'
                       database-id=enu database-version=aurora2
```

and the resulting *STRUT* segmentation files are merged into an archive file once the `*.08` files have been removed,

```
> create-archive output=segmentation/train.seg format=samples
                 dir=segmentation/train
```

With the first segmentation, we can train a first acoustic model as described in the previous sections.

The acoustic model can be refined by realigning the segmentation. To do so, we use `recognize` itself. Indeed, if you recognize a speech utterance with a lexicon containing only the pronounced words and a grammar limited to the pronounced sentence, the recognition is straightforward and a byproduct of the recognition will be the state segmentation. This procedure is generally called *forced alignment*.

First, we build the phonetic file. This file contains a special application file for every utterance to align which allows only the actual sentence. Such a file can be build with the perl/Tk interface. Select the "Compile" page. Then, you can compute alternatively a new segmentation and a new acoustic model. See "Training Perl Scripts".

Chapter 3

Testing Procedure

Create a Sample File

Like the training procedure, the testing procedure begins with creating a archive file with the test utterances. First, we copy the 1001 test files `speechdata/testa/clean1/*.08` from cdrom 1/4 of the Aurora2 collection into the `/samples/test` directory. Then, the files are strutified, for example:

```
> strutify input=samples/test/fak_1b.08
           output=\DATADIR/samples/test/fak_1b
           data-format=big-endian data-size=2
           file-type=samples'(sample-rate=8000)'
           database-id=enu database-version=aurora2
```

and the archive file is obtained once we have removed the `*.08` files:

```
> create-archive output=samples/test.sam format=samples
                 dir=samples/test
```

This file can be directly used as input for the decoding program `recognize`, yet you can compute the features beforehand to save computational time during the decoding process.

Create a Language Model

The next step of the testing procedure is to create the language models in the form of a FSG file. Figure 1.5 gives the FSG file for recognizing English digit sequences including a garbage model. This file has typically the file extension `*.fsg` and is located in the `application` subdirectory. In our example, we name it `aurora2.fsg`.

Compile an Application File

As we mentioned in section 1.3.4, the FSG file and the lexicon are compiled to obtain an application file to be used by the decoding program `recognize`. This is done with the application compiler `compile-asr`:

```
> compile-asr phonemes=application/aurora2-words.hmm
              user-dictionary=application/aurora2-words.dic
```

```
                mode=fsg'(syntax=application/aurora2.fsg)'
                output=application/aurora2-words.app
```

Based on the HMM topology file `aurora2-words.hmm` given in Appendix C.0.2, the lexicon `aurora2-words.dic` of figure 1.4(a) and the FSG file `aurora2.fsg`, all located in the `application` directory, `compiler-asr` builds a state graph which defines all the allowable state sequences. The application file has typically the `*.app` extension and it is located in the `application` directory. Likewise, the application file for phoneme-based HMMs is generated as follows:

```
> compile-asr phonemes=application/aurora2-phonemes.hmm
              user-dictionary=application/aurora2-phonemes.dic
              mode=fsg'(syntax=application/aurora2.fsg)'
              output=application/aurora2-phonemes.app
```

with the HMM topology file `aurora2-phonemes.hmm` and the lexicon `aurora2-phonemes.dic` given in Appendix C.0.2 and figure 1.4(b), respectively, and the same FSG file `aurora2.fsg`.

Recognize a Sample File

Finally, we can recognize the test set with the program `recognize`. This program decodes a set of test utterances with respect to an application file and a MLP. For example, we can recognize the test set `test.sam` with the ASR system based on MEL acoustic features and phoneme-based HMMs:

```
> recognize input=samples/test.sam
            output-type=words output=results/test-phonemes-mel.hyp
            models=yes'(file=models/enu08.ff.234-0600-033.mel.mlp)'
            decode=yes'(application=application/aurora2-phonemes.app)
```

The results of the recognition process are contained in the hypothesis file `test-phonemes-mel.hyp`. An hypothesis file has typically the `*.hyp` extension and is located in the `results` subdirectory.

Assess System Performance

In order to assess an ASR system, it is common to compute statistics on its recognition performance. To do so, we compare an hypothesis file (what has been recognized) with a reference file (what should be recognized) and count the number of errors. Figure 3.1 shows some lines of the hypothesis `test-phonemes-mel.hyp` and Figure 3.2 the corresponding lines of the reference file `test.ref`. The format of both hypothesis files ans reference files is simple: word string (utterance identifier). The reference file has typically the `*.ref` extension and it is located in `reference`.

Three types of errors are classically considered: substitution, deletion and insertion errors. The Perl script `sclite-score.pl` allows to compute such statistics. First, it aligns every recognized word string from the hypothesis file to the corresponding word string from the reference file. Then, it counts the errors and derives error rates as well as confidence intervals:

```
> sclite-score.pl -r reference/test.ref
                  -h results/test-phonemes-mel.hyp
                  -M 01
```

Acoustic Features	HMM type	
	word-based	phoneme-based
MEL	0.83% (0.31% / 0.18% / 0.34%)	1.83% (0.64% / 0.46% / 0.73%)
PLP	0.80% (0.34% / 0.09% / 0.37%)	1.59% (0.61% / 0.46% / 0.52%)

Table 3.1: Word error rate (substitution/deletion/insertion).

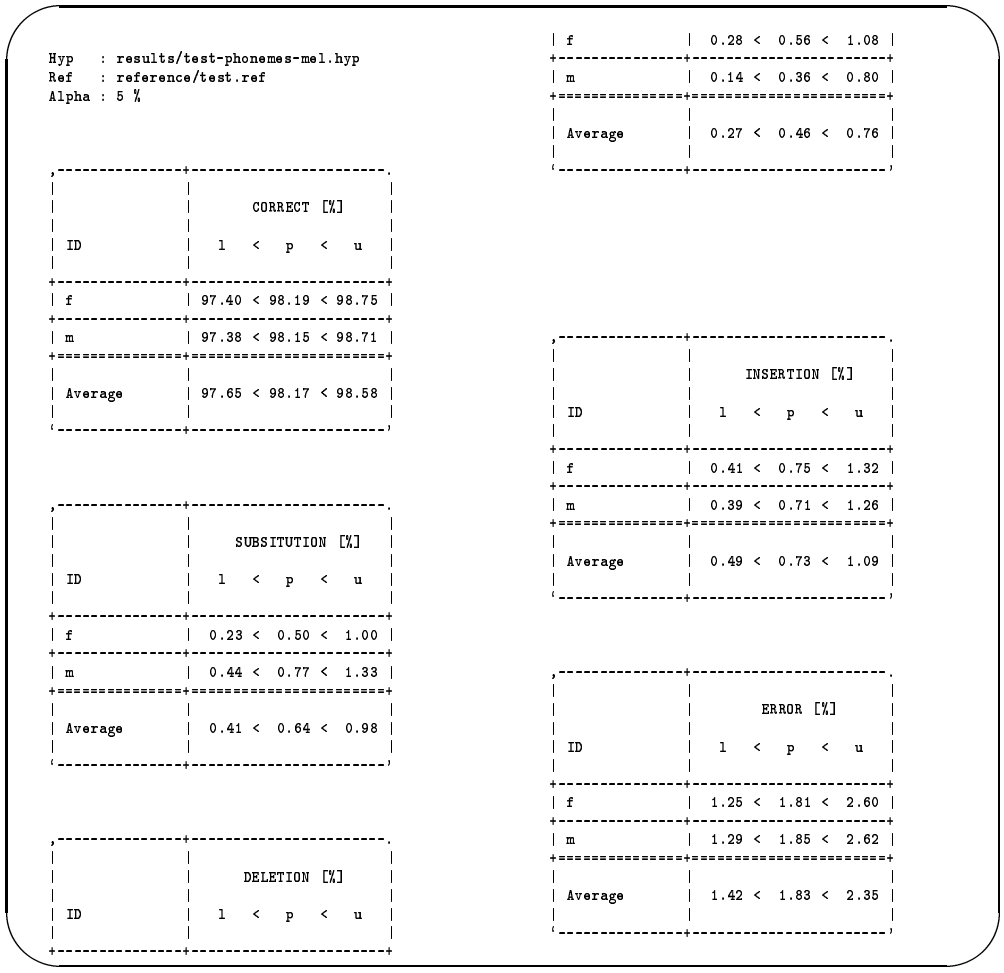
```
...
five five four (fbl_854a)
nine six oh (fgb_96ooa)
two seven five oh (fba_27oa)
five three three (mle_933a)
four one (mjh_419a)
two oh six eight (mhm_268a)
...
```

Figure 3.1: The hypothesis file `test-phoneme-mel.hyp`

Thanks to a mask option (`-M 01`), it is possible to filter the utterance ids and to provide statistics by group of utterances. In our example, we use the first letter of the utterance ids (the left parenthesis is ignored) to group the utterances, separating female speakers from male speakers. Figure 3.3 gives the output of the above command. It shows that the ASR system based on MEL acoustic features and phoneme-based HMMs works fairly well with a word error rate equal to 1.83%. For the sake of comparison, we also give the word error rates for the ASR systems based on PLP acoustic features and word-based HMMs in table 3.1.

```
...
five five four (fbl_854a)
nine six oh (fgb_96ooa)
two seven five oh (fba_27oa)
five three three (mle_933a)
four one (mjh_419a)
two oh six eight (mhm_268a)
...
```

Figure 3.2: The reference file `test.ref`



```
Hyp   : results/test-phonemes-mel.hyp
Ref   : reference/test.ref
Alpha : 5 %
```

ID	CORRECT [%] l < p < u
f	97.40 < 98.19 < 98.75
m	97.38 < 98.15 < 98.71
Average	97.65 < 98.17 < 98.58

ID	SUBSITUTION [%] l < p < u
f	0.23 < 0.50 < 1.00
m	0.44 < 0.77 < 1.33
Average	0.41 < 0.64 < 0.98

ID	DELETION [%] l < p < u
f	0.28 < 0.56 < 1.08
m	0.14 < 0.36 < 0.80
Average	0.27 < 0.46 < 0.76

ID	INSERTION [%] l < p < u
f	0.41 < 0.75 < 1.32
m	0.39 < 0.71 < 1.26
Average	0.49 < 0.73 < 1.09

ID	ERROR [%] l < p < u
f	1.25 < 1.81 < 2.60
m	1.29 < 1.85 < 2.62
Average	1.42 < 1.83 < 2.35

Figure 3.3: ASR Results.

Chapter 4

The Tools

This chapter contains a description of the tools that will allow to train models, and test recognition.

4.1 Database Handling

4.1.1 General Purpose

strutify raw to *STRUT* conversion tool. The main task is to add a *STRUT* header, where the user can specify the fields and the values. The program is able to skip a fixed-length header.

edit-header allows to add/remove/edit fields in the header of a *STRUT* file.

create-archive reads *STRUT* files (note that a *'wav'* file can be considered as a *STRUT* file) in a [list of] directories, and pack all the data in a single file, creating a *STRUT* archive. It understands the data that it reads, so it is able to optionaly code the data. For instance, the default behaviour when it handles samples, is to code them into the *SHORTEN* format.

unstrutify takes a *STRUT* archive and split the data into one file per utterance

select-utterance allows to create an archive that is a subset of another one. Note that the *STRUT* programs are normally able to do it on the fly.

StrutSurfer derives from *WaveSurfer*. Please see chapter 5 for details.

4.1.2 Samples

Some programs allow to manipulate a sample database:

convert-samples is able to modify the sampling rate, modify the sample format, and so on

add-noise takes a sample file and adds random or database noise, to get a given sample-to-noise ratio.

wiener is supposed to enhance the speech quality, by spectral filtering.

4.2 Training and Testing of Models

compile-asr takes an input vocabulary and/or grammar, together with a phonetizer, and creates an application file for the recognizer. A special format of the output allows to keep backtracking information for the phonemes, thus allowing to turn the recognition into a forced alignment.

recognize reads data in a given format: samples, features, probabilities, and writes them into another format: features, probabailitiues, or words. It is also able to provide a phoneme segmentation.

align is a script that repeatedly calls recognize to produce a segmentation of a database, and post process that segmentation into a suitable format for mlp-train.

train.ptk is a Perl/Tk script that trains an mlp. It calls the following 3 programs:

mlp-init computes the feature normnalization parameters.

mlp-train performs one iteration of the mlp traning.

mlp-cross-validate does a cross-validation on a test set, to evaluate the performance of the MLP.

Chapter 5

StrutSurfer

StrutSurfer is a sound edition tool based on KTH's *WaveSurfer*, dedicated to the STRUT toolkit. The user is invited to read the *WaveSurfer* documentation for more information. *StrutSurfer* can not only view, edit, and play a huge variety of sound files (including *STRUT* files and archives), but can also be used to compute and visualize features and/or probabilities, segmentations, alignments, and much more.

5.1 The *StrutSurfer* Window

The *StrutSurfer* window looks much like WaveSurfer window. Based on *WaveSurfer*, it inherits from all WaveSurfer facilities and mouse gestures.

5.2 Basic Functions

5.2.1 The *File* Menu

The *File* menu contains the usual functions:

Open to open a sample file

New to create a new window

Save to save the current utterance

Save as to save the current utterance

Quit If you want that operation now, you can skip this chapter

Additionally, the **File** menu contains the 5-most recent files, and a *File Associations* entry, that allows the user to define custom file types, based on file extension. When *StrutSurfer* encounters an unknown file type, it asks the user information about the sampling frequency, sample coding, etc... The user may choose to associate the file extension to those settings. When the the *File Associations* button is clicked, a window displaying the currently associated extensions pops up, and provides the possibility to remove any of these extensions.

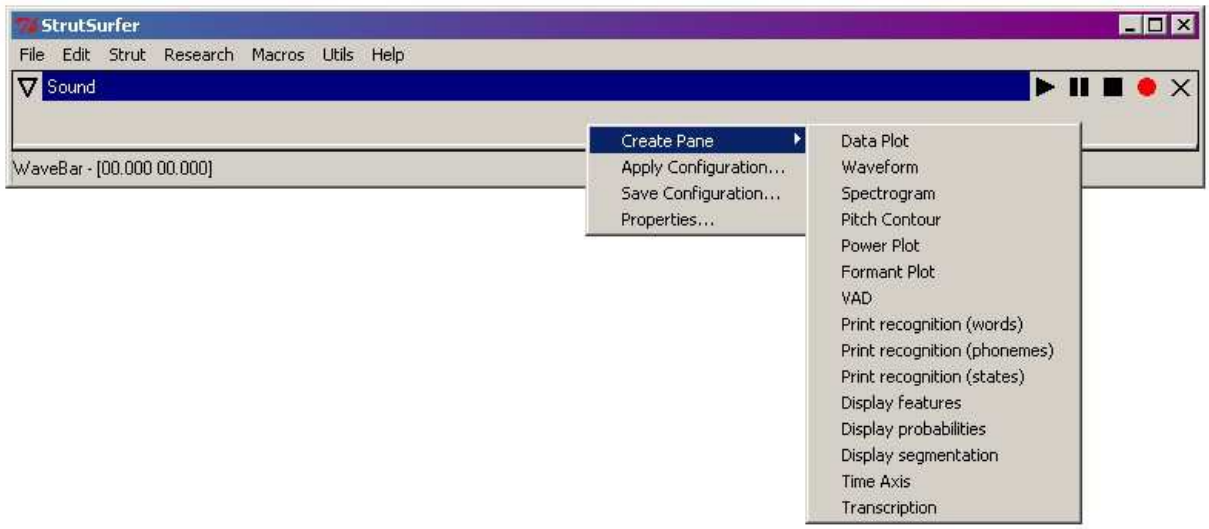


Figure 5.1: A *StrutSurfer* Window

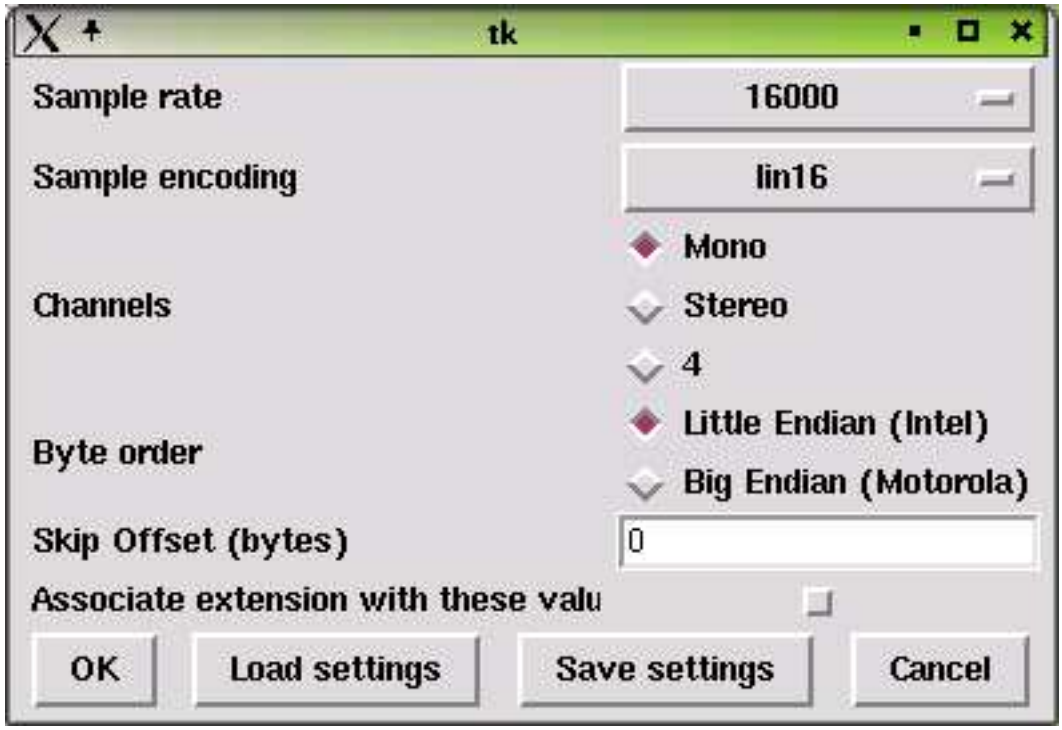


Figure 5.2: The *File Associations* setting panel

Figure 5.3: The *File Associations* management panel

Opening a *STRUT* Archive

When *StrutSurfer* opens a *STRUT* archive, a window pops up, with the list of all available utterances. The user can select an utterance by simply double-clicking on the utterance ID, or by typing it (fig 5.4).

5.2.2 The *Edit* Menu

The **Edit** menu contains the following commands:

Undo and **Redo** to undo/redo the last command

Copy to copy the selected region

Cut to cut the selection

Paste to paste the contents of the clipboard

Add to to ad samples in the clipboard to te current sound. This can be used, for instance, to add noise to a signal (fig. 5.5).

Previous Utterance to switch to the previous utterance in the archive.

Next Utterance to switch to the next utterance in the archive.

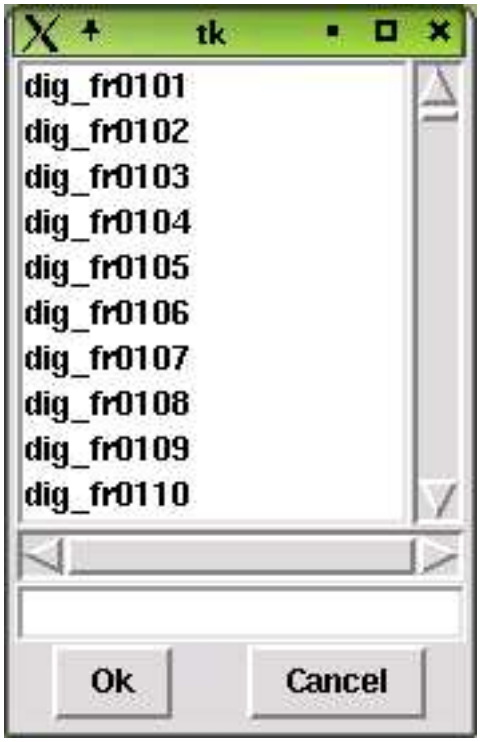


Figure 5.4: A small panel displaying the utterance ids of a *STRUT* archive

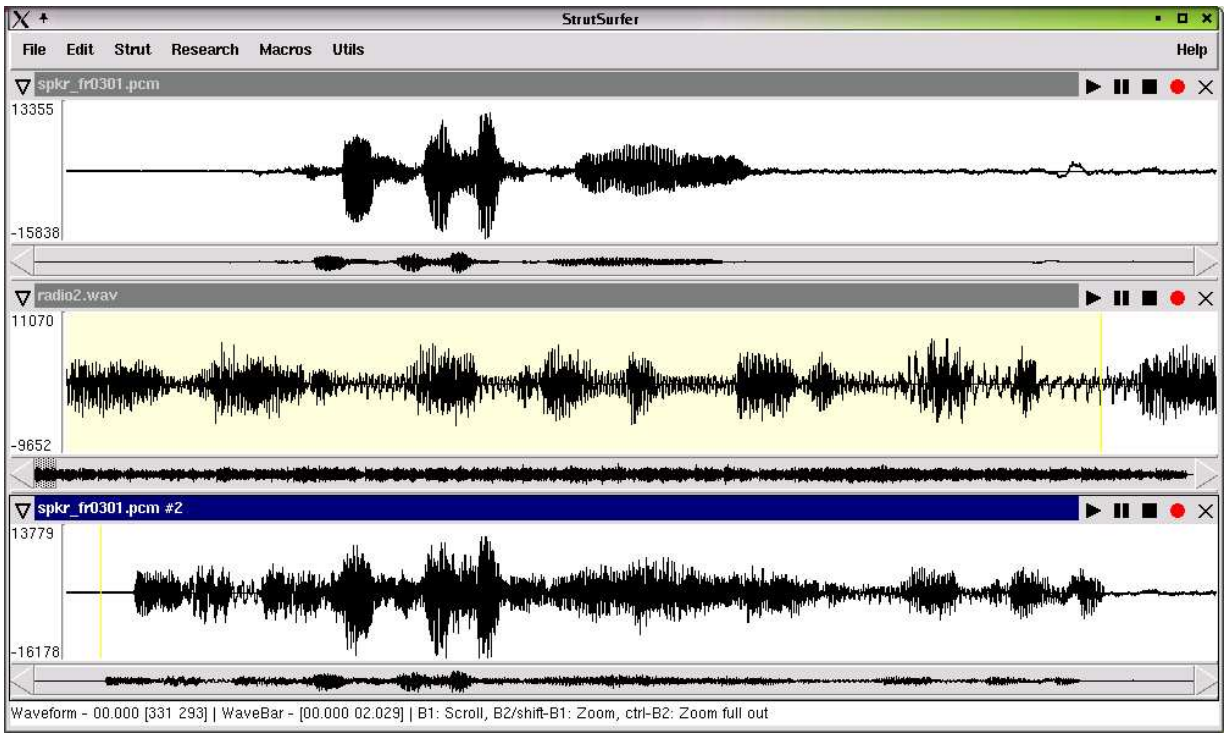


Figure 5.5: Adding the samples in the clipboard to an utterance

5.3 ASR Functions

5.3.1 The *Strut* Menu

The **Strut** menu provides basic ASR functionalities. The user can open and set up a recognizer, choose acoustic models, applications, and perform a recognition:

New Recognizer creates a new recognizer object with default parameters, without models and application.

Open Language will let you choose acoustic models and assigne the to the current (selected) recognizer.

Open Application is not active if you did not open a language model. The command allow to assign an application to the current recognizer).

Recognize Once a recognizer is properly set (it has been assigned at least a language and an application), the command is enabled, and it is possible to perform recognition.

Recognizer Settings allow to change the parameters of the current recognizer.

5.4 The *Utils* Menu

Besides simply recognizing a buffer of samples, *StrutSurfer* can be used to manipulate the sound, i.e. adding different types of noise, filtering or enhancing the speech signal. All these features are accessible via the **Utils** menu. The menu currently contains 3 submenus, one for each operation (adding noise, filtering or enhancing speech), each submenu being composed of two commands: the operation itself, and the setting of the operation parameters. As an example, we show below a screenshot of the *Noise settings* command, which controls the parameters of additive noise.

5.5 Advanced Features

In this section, we review the advanced features (features in **Research** menu, and provided by specific plugins).

5.5.1 Computing Features and Probabilities

StrutSurfer can easily compute features and probabilities, as soon as a recognizer is provided with acoustic models and an application. This allows in particular to compare a reference stream to an online-computed stream (to test changes in the code, or simply the results returned by two models), or to detect phonemes that are abnormally recognized, leading to bad recognition results.

To compute features or probabilities, the user must provide a recognizer with acoustic models and an application, and then execute the appropriate command in the **Research** menu. Once the features or probabilities have been computed, the streams can be visualized by creating a new pane (fig 5.8).

It is also possible to load features or probabilities from a file. The file must contain the utterance ID corresponding to the current sound, if this sound comes from a STRUT archive, or

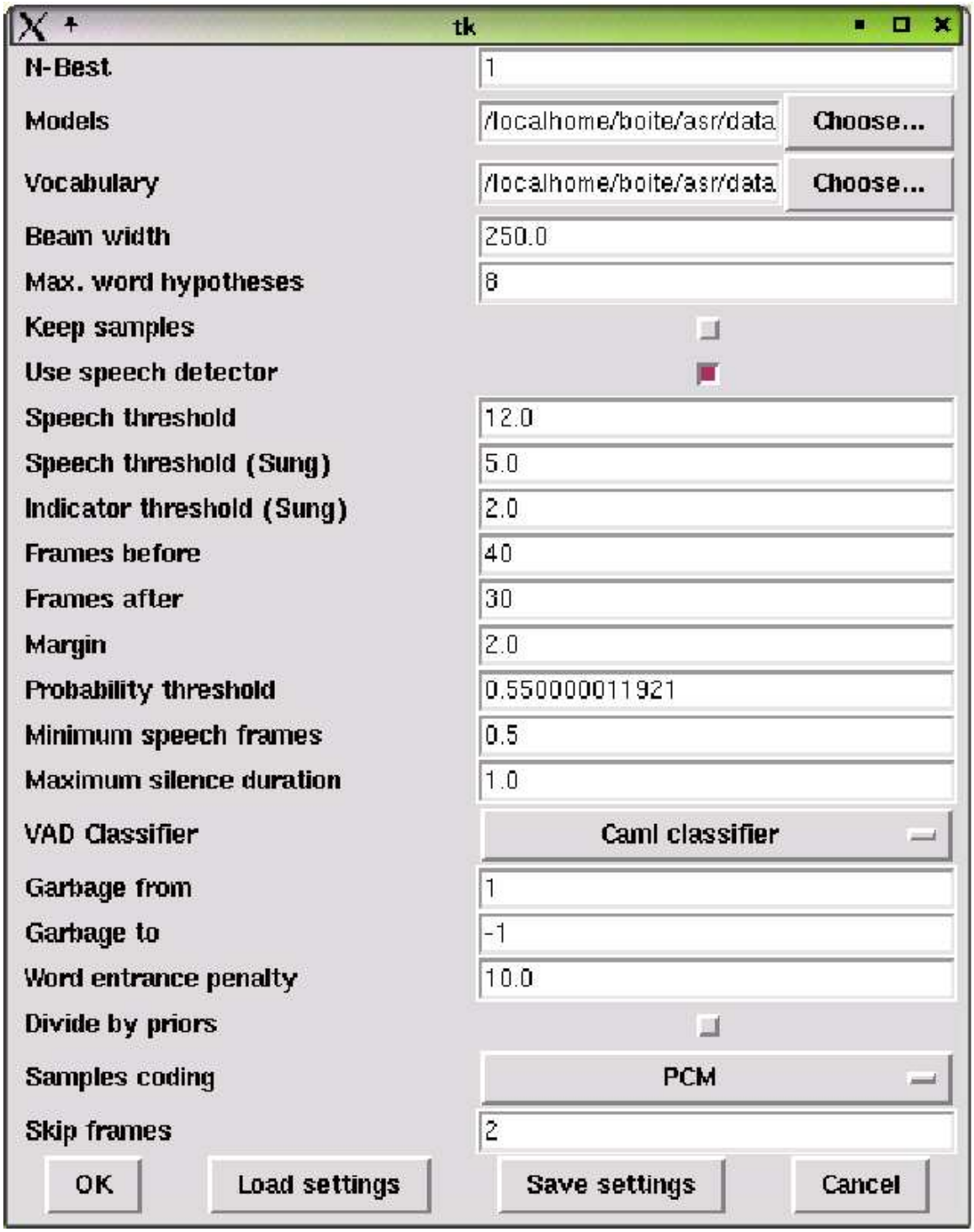


Figure 5.6: Setting the parameters of the recognizer

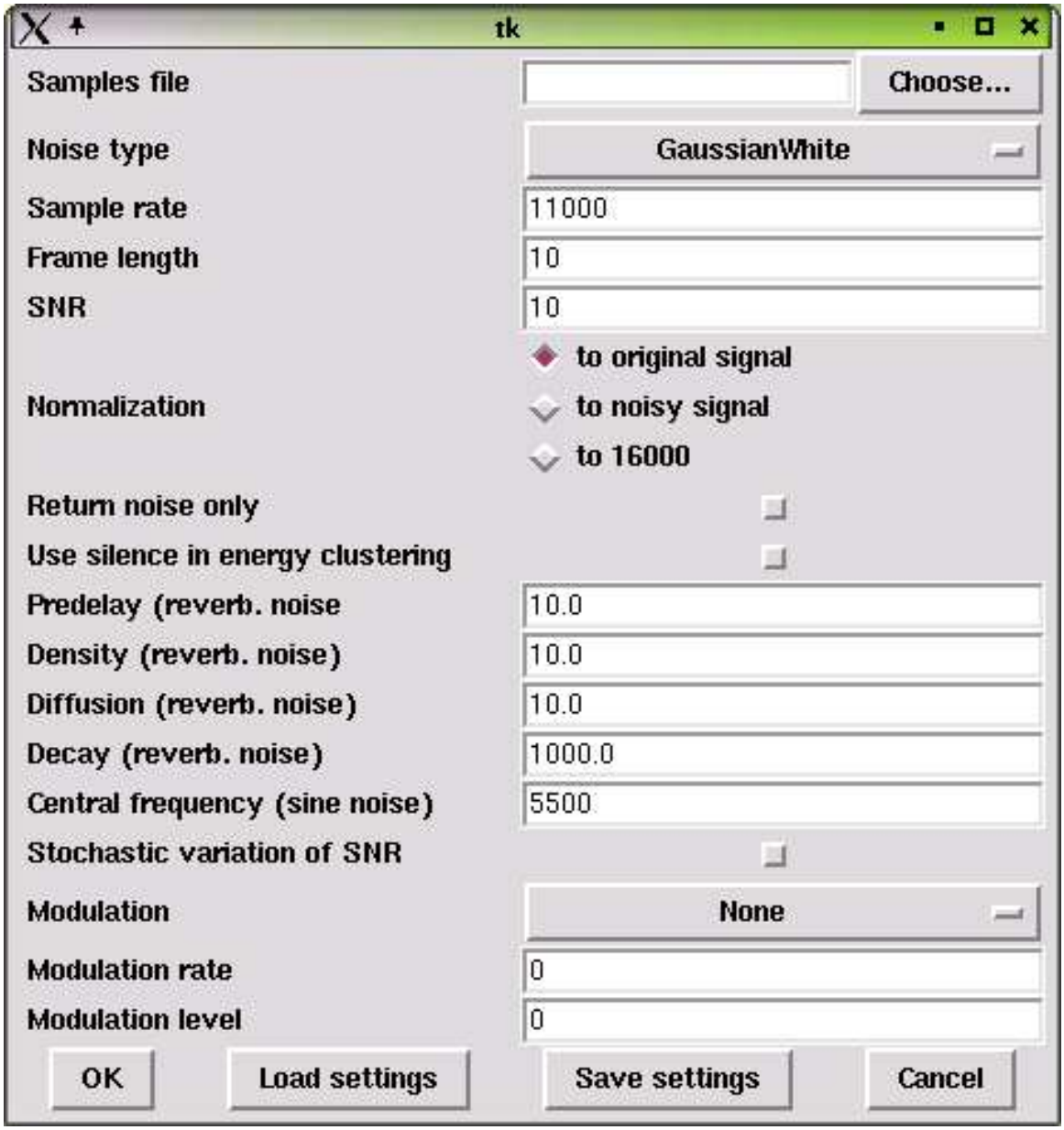


Figure 5.7: Setting the characteristics of the noise to be added

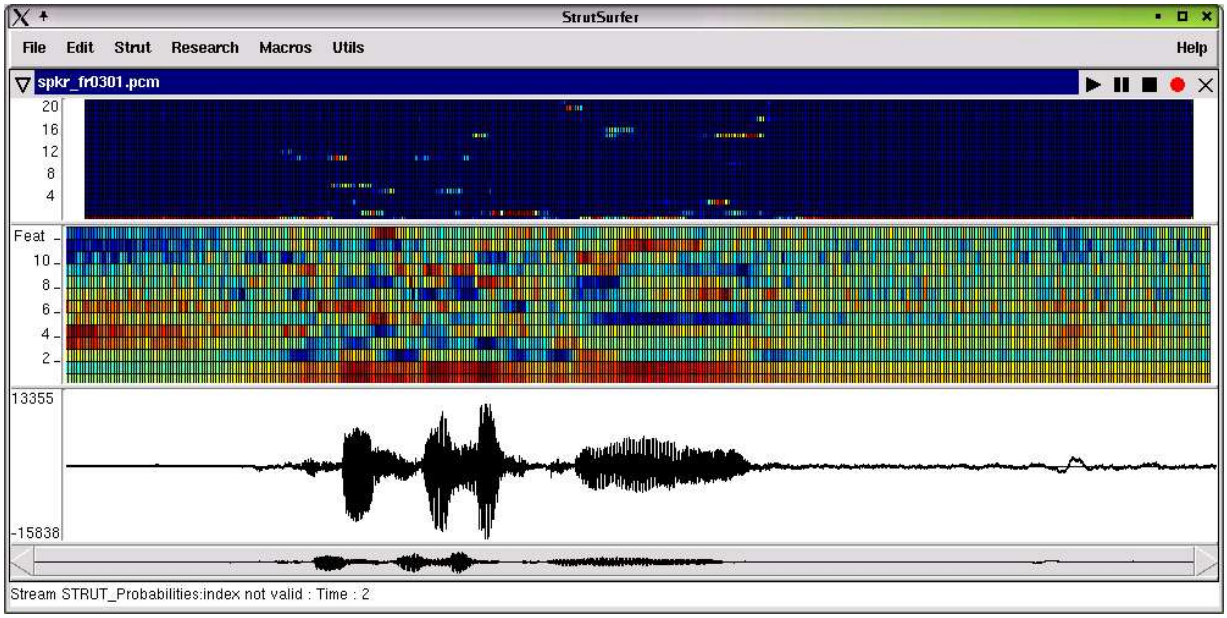


Figure 5.8: Displaying the features and the probabilities computed by the *MLP*

contain a single utterance, if the current sound has no utterance ID (i.e. was not extracted from an archive).

In case the current sound has been extracted from an archive of samples, and a corresponding archive of *features* (resp. *probabilities*) exists and respects the `/asr/database` directory structure, then this archive is automatically loaded.

The `/asr/database` directory structure is described in chapter A.2.1.

It is also possible to open a segmentation file corresponding to an utterance. If the utterance is part of a *STRUT* archive, the segmentation file is automatically loaded, as described above. In other cases, or if the user wants to specify an alternate segmentation file, he/she just has to open a segmentation pane, and choose a file by right clicking into this pane.

5.5.2 Performing Alignment

It is fairly easy to perform an alignment (or any kind of alternate recognition) with *StrutSurfer*. The user only needs to open a segmentation pane, and load an application. StrutSurfer then automatically generates a recognizer with the same settings as the current recognizer (except for the application, which is the application the user just loaded), and performs the recognition. The recognition results are then displayed into the segmentation pane.

Note that you need to have acoustic models associated to your recognizer. If it is not the case, StrutSurfer warns you.

By default, if several segmentations are available (words, phonemes and/or states, controlled by the application), the most detailed one is used.

You can also perform an alignment using an archive of applications. As usual, in this case, *StrutSurfer* looks for the current utterance ID, if any. If the sound buffer was not extracted from an archive, then the archive of application must contain only one utterance. If the correct utterance cannot be found, *StrutSurfer* silently returns.

5.6 Macros

StrutSurfer provides a macro mechanism, so that the user does not have to continuously repeat the same operations. Combined with the configurations provided by the underlying *WaveSurfer* layer, it makes it very fast and easy to perform the same task on several files/utterances.

To use macros, one just has to click on the **Record a macro** command, execute normally the sequence of tasks to be recorded into the macro, and finish recording the macro by clicking on the **Stop recording macro** command.

The user must pay attention, however, to several things:

- Sound buffer manipulations, such as **paste**, **cut** or **add to** 5.2.2 involve a buffer of samples and a selection (cursor position) that are ALSO recorded into the macro.
- **Undo** and **Redo** commands only undo/redo the last operation. Be careful that undo/redo operations executed BEFORE the recording of a macro are not taken into account.
- **Previous Utterance** and **Next Utterance** may not change utterance, in case the first (resp. the last) utterance is selected.

Once the user has finished recording a macro, he/she has to register it and give it a name. The name then appears in the **Macros** menu, and the macro can be called by clicking on its name. When *StrutSurfer* is closed, macros are written into a file, so that they remain available through sessions.

Part II

STRUT Reference Guide

Chapter 6

The Strut User Interface

6.1 Introduction

The general syntax of a command is: `command parameter=value ...`

For all programs, typing `command help=yes`, of for almost all of them (those which have mandatory arguments), typing the command without any parameters gives an explanation on the command parameters. On-line help is also available in html format. There is also a *TCL/TK* interface, but it is in a very preliminary state.

There is also another form of a command, which is obsolete and should not be used any more:

```
command database_id/database_version parameter=value ...
```

STRUT used to be a collection of different programs that performed dedicated actions. this is not the case any more. *STRUT* is now a single program, with all the functionality. This is transparent for the user on Unix-like systems, where you have symbolic links. The program reads its first argument (the program name) to know which functionality it is supposed to implement, and how it is supposed to parse the remaining arguments. If you don't have symbolic links, you can always directly call the main program with the name of the component as the very first argument: `strut recognize setup=recognize.stp output=-`.

6.2 Case Dependency and Shell Interaction

There is no distinction between lower case and upper case letters in arguments names. Also, there is no distinction between dashes (-) and underscores (_).

To allow filename completion, and wildcard expansion, any number of space character can be put between the equal sign and the value of the parameter: `extract-features  setup= lpc-cepstrum` is valid, but not: `extract-features  setup =lpc-cepstrum`.

After all the parameters can follow a list of filenames. That's why command line is not parsed beyond the parameter with no equal sign.

6.3 Argument Types

This section will describe all the argument types that have been defined. Almost all of them can be arrays, or arrays of arrays. If you need to enter an array, you can either:

- Enclose the values between [] or , and separate values with commas or spaces (commas are only valid for numeric parameters):

```
float-array-param=[0.621 1.4142136 2.7182818 3.1415927]
```

  When you specify int, float, or bool params, the parenthesis are there for clarity only. If you separate values by commas, you don't need them:

```
float-array-param=0.621,1.4142136,2.7182818,3.1415927
```

- When specifying string array, include the strings between quotes:

```
string-array-param=["s1", "s2"]
```

- Specify the individual values, alone or grouped:

```
float-array-param[0]=.621
float-array-param[1]=1.4142136
float-array-param[2..23]=2.7182818
float-array-param[24]=3.1415927
float-array-param[25..27]=3.1415927
```

- Some hybrid notations are also available, but without guaranty:

```
float-array-param=0.621,1.4142136,2.7182818,3.1415927
float-array-param[1]=0.71
```

  The rule is simple:

```
float-array-param[size]=28
```

  and

```
float-array-param=0.621,1.4142136,2.7182818,3.1415927
```

  both specify the size of the array. And construct like:

```
float-array-param[1]=0.71
```

  modify one element.

  To enter arrays of arrays, the allowed syntaxes are similar:

```
float-array-array-param=[[2 3 7][2 3][3 4 5 6 7 8]]
float-array-array-param[2][0]=3
float-array-array-param[0..2][1]=9
float-array-array-param[0..2][1..6]=96
float-array-array-param[0:2][3:4]=3
float-array-array-param[0]=[2 3 7]
```

Of course, using [] and can cause problem with your shell, so array values will preferably specified in setup files (see section 6.4).

Values are:

Booleans The boolean will be set to true if the param is specified as "1.*", "[tT].*" or "[yY].*", to false otherwise. All input values are valid.

Integers They can be restricted to be only positive, or only negative, or min and max values can be specified.

Floats They can be restricted to be only positive, or only negative, or to be between 0 and 1.

Strings Any ASCII string. Don't forget to put the string between quotes if there is a space or any special character interpreted by your shell.

Filenames filenames required can be input or output files.

Keyword Only a keyword is allowed

Menu The menus introduce a hierarchy in the command line. They can take some keyword values, and according to the keyword chosen, introduce a new parameter list, specified between parenthesis:

```
command db/dbid foo=bar(par1=yes par2=toto)
```

Or:

```
command db/dbid foo=quick(par3=no)
```

Often, those menus are introduced only for clarity, and no keyword is associated with them:

```
command db/dbid foo=(par1=yes par2=toto)
```

The full syntax of the menu is not displayed. To get help, specify help=yes as parameter:

```
command foo=(help=yes)
```

6.4 Common Parameters

Besides the parameters individually defined for each command, a few parameters are common:

help the help parameters allow to get help on a command or a general help on the command line parsing. The value is a keyword in the list:

help print this text

y(es) print help on the command (or the sub menu)

all print all help available for this command (current level plus sub menus).

latex print the help in latex format

html print the help in html format

tcl creates and runs a TCL/TK script to ease the specification of the parameters. You need TCL/TK installed on your system.

data-dir-variable print the name of the environment variable which tells where the partially specified files are to be found.

data-dir the contents of the `data-dir-variable`

verbose This menu controls the setting of some debugging variables:

oobp the more this integer is, the more verbose is the oobp mechanism

memory print information about the memory in use by the program

stream-header print debugging information about what is found in the header of the files.

print-params print parameter values after command line parsing.

setup apparition of such an argument causes the program to read the filename specified. The file is supposed to contain arguments to the program, with the same syntax as in the command line, except that empty lines or lines beginning with # are ignored. `setup` files can be nested to any level. For good readability, long lines can be split: simply put a backslash at the end of the line.

setenv This can be used to store text in variables, for later use, à la `pmake`. The value of a variable may be retrieved by enclosing the variable name in parenthesis and preceding the whole with a dollar sign. For example, you can put in your setup:

```
setenv PHONEMES /home/strut/data/phonemes/english
phonemes= $(PHONEMES)
```

Variables used inside another variable are expanded whenever the outer variable is expanded. So, in the example:

```
setenv STRUT_DIR  /home/strut
setenv PHONEMES $(STRUT_DIR)/data/phonemes/english
first= $(PHONEMES)
setenv STRUT_DIR  /home/zorglub
second= $(PHONEMES)
```

`first` will receive the value `/home/strut/data/phonemes/english`, while `second` will receive `/home/zorglub/data/phonemes/english`.

- The variable values can be put in the environment, prior to execution of the program, so you can type in your shell:

  ```
  setenv MYDATABASE digits/1.0
  ```

  or

  ```
  export MYDATABASE digits/1.0
  ```

  and put in your setup:

  ```
  weights $(STRUT_DATA_DIR)/$(MYDATABASE)/weights.130-200-66
  ```

  Three variables are already defined, if you specify the database id in the command line:

  DATABASE_ID
  DATABASE_VERSION
  DATABASE , which is `DATABASE_ID/DATABASE_VERSION`

  If a parameter has a default value, it can contain variables. The default value are expanded the command line and the setup files are read, so when variables are set.

6.5 Multiple Specification of an Argument

You you specify an argument that you have already specified, the value is simply overwritten.

When you specify menus, for reason of clarity in setup files, you can split in several lines:

```
weights=(file=afac)
weights=(scratch=yes)
```

The values you specified are not reset, unless you specify a different keyword to introduce the menu:

If, after saying:

```
menu=read(file=null type=ascii)
```

You say:

```
menu=write(type=ascii)
```

The file argument has been reset (as it may not be relevant for the "write" menu).

6.6 Command Line Help

The program that will be presented here is "recognize", which has been choosen because it has many options. The operating system used here is Linux, but it works *mutatis mutandis* under Windows or MacOS.

General Help If you type "recognize help=yes", or simply "recognize", you will get something like in figure 6.1.

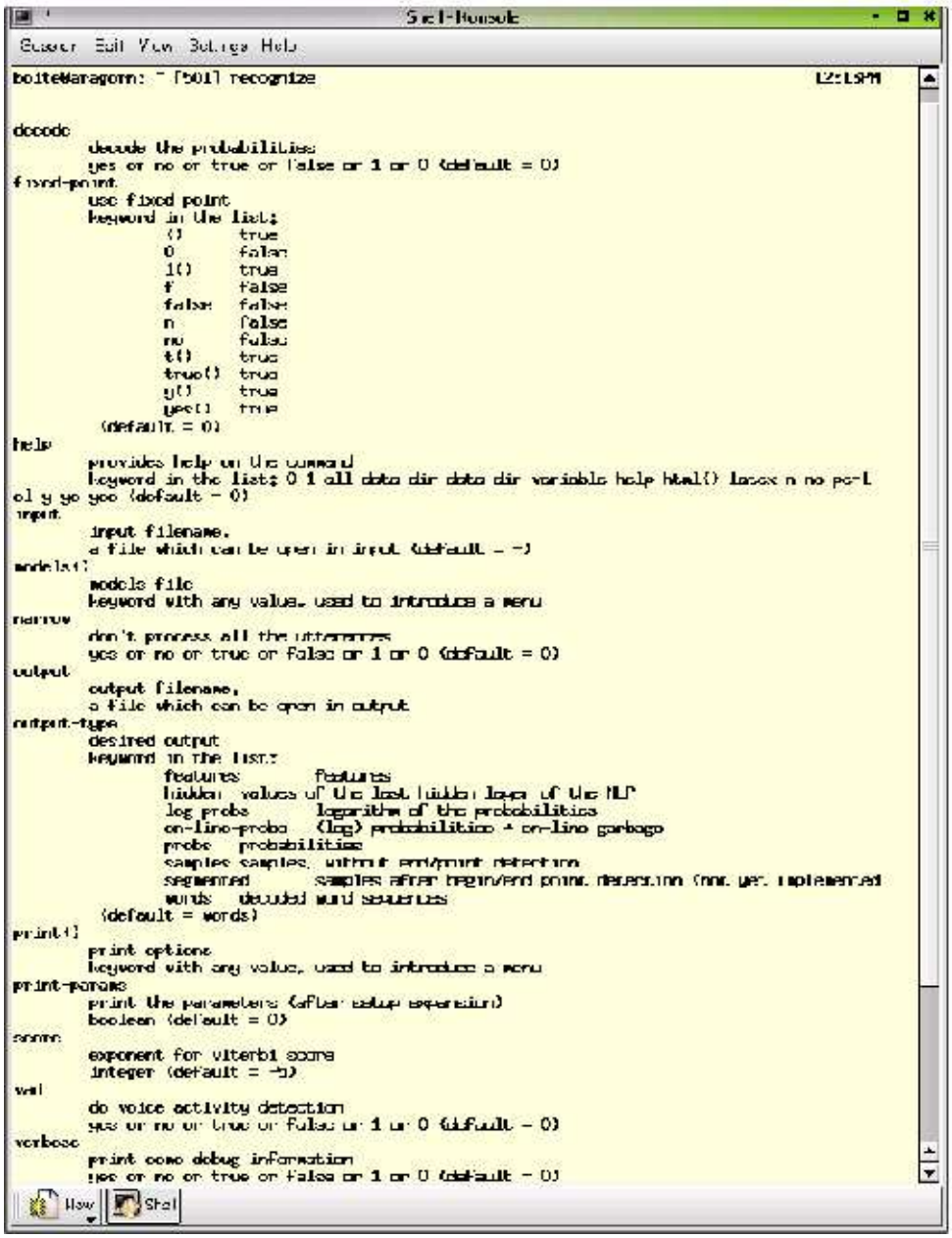

Shell-Konsole
Session Edit View Bookmarks Help
bolteNaragorn: ~ [501] recognize
12:15PM
decode
decode the probabilities
yes or no or true or false or 1 or 0 (default = 0)
fixed-point
use fixed point
keyword in the list:
(1 true
0 false
1() true
f false
false false
n false
no false
t() true
true() true
y() true
yes() true
(default = 0)
help
provides help on the command
keyword in the list: 0 1 all data dir data dir variable help html() lesox n no pc-l
ol y yo yoo (default = 0)
input
input filename.
a file which can be open in input (default = -)
models()
models file
keyword with any value, used to introduce a menu
narrow
don't process all the utterances
yes or no or true or false or 1 or 0 (default = 0)
output
output filename.
a file which can be open in output
output-type
desired output
keyword in the list:
features features
hidden values of the last hidden layer of the MLP
log probs logarithm of the probabilities
on-line-probs (log) probabilities + on-line garbage
probs probabilities
samples samples, without end/print detection
segmented samples after begin/end point detection (not yet implemented)
words decoded word sequences
(default = words)
print()
print options
keyword with any value, used to introduce a menu
print-params
print the parameters (after setup expansion)
boolean (default = 0)
scomp
exponent for viterbi score
integer (default = -5)
vad
do voice activity detection
yes or no or true or false or 1 or 0 (default = 0)
verbose
print some debug information
yes or no or true or false or 1 or 0 (default = 0)
New
Shell

Figure 6.1: General Help

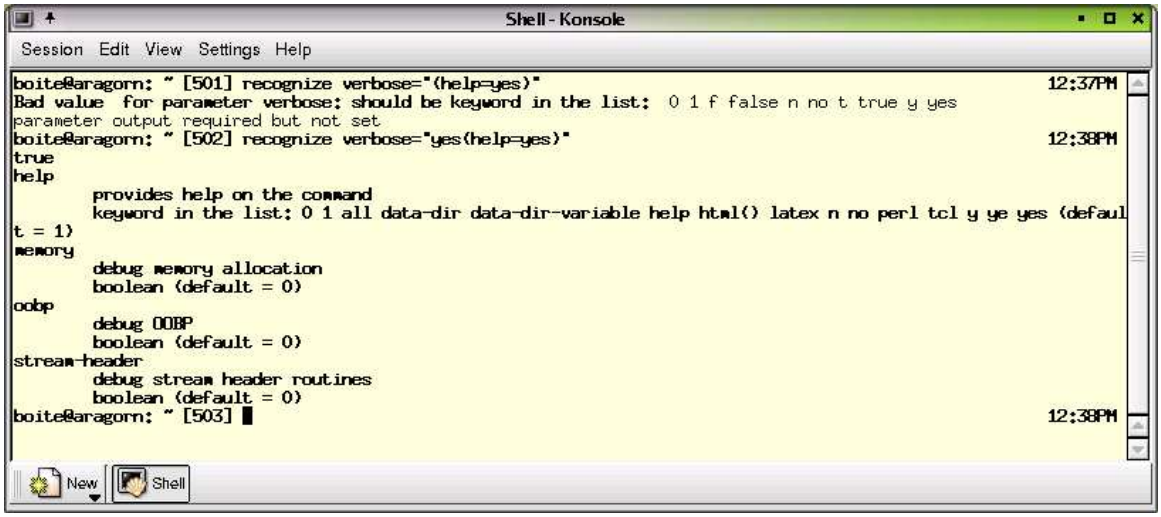

Shell - Konsole
Session Edit View Settings Help
boite@aragorn: " [501] recognize verbose="(help=yes)" 12:37PM
Bad value for parameter verbose: should be keyword in the list: 0 1 f false n no t true y yes
parameter output required but not set
boite@aragorn: " [502] recognize verbose="yes(help=yes)" 12:38PM
true
help
provides help on the command
keyword in the list: 0 1 all data-dir data-dir-variable help html() latex n no perl tcl y ye yes (defaul
t = 1)
memory
debug memory allocation
boolean (default = 0)
oobp
debug OOBP
boolean (default = 0)
stream-header
debug stream header routines
boolean (default = 0)
boite@aragorn: " [503] 12:38PM
New
Shell

Figure 6.2: Help on the verbose menu

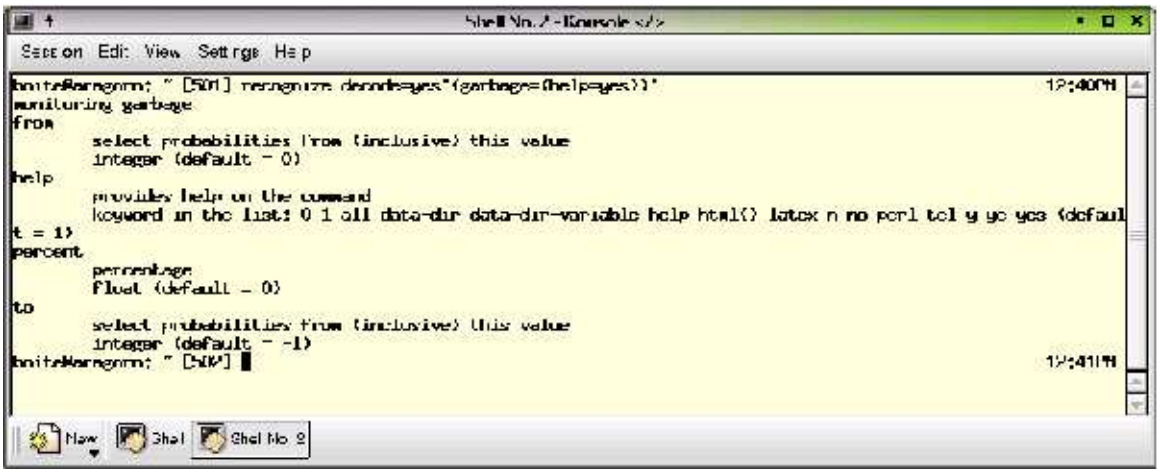


Figure 6.3: Help on the garbage sub-menu

Detailed Menu If a parameter has a menu, you can get help about the menu with a command like "recognize verbose=yes(help=yes)". As the menu depends on the parameter value, you have to specify a value (see fig 6.2).

Nested Menus Sometimes, menu will be nested, like in fig 6.3.

Chapter 7

The Files

STRUT has defined its own database format, inspired by the ***NIST*** *SPhere* wave file format: an ASCII header describes what's in the file, and the data follows, in binary format. Extensions and adaptations have bee made to the format.

STRUT is able to handle:

- *SPHERE* headered files. These are samples files, with a ASCII header. Each file contains one utterance. Many speech databases follow that format. A header is a succession of fields that describes the data format or that gives any information on how the data has been obtained. Each line consists of the field name, followed by a dash letter specifying the type of the value, and then the field value itself:

  -i integer

  -s23 string of 23 characters

  -f float

- *Micro$oft wave* file format.

- *STRUT* file format. A single file can contain a complete database. The header is similar to the *NIST* header, with extension to arrays and arrays of arrays, like in:

```
mlp_cross_validation_score -f 85.2903 86.1208 87.2948
feature_dimension -i 13
feature_multipliers[0] -f 0 0 1 1 1 1 1 1 1 1 1 1 1
feature_multipliers[1] -f 1 1 1 1 1 1 1 1 1 1 1 1 1
feature_multipliers[2] -f 1 0 0 0 0 0 0 0 0 0 0 0 0
```

7.1 Strut File Headers

In a *STRUT* header, no field is mandatory, and there is no restriction on file names. However, some fields have a special meaning, and can affect the way *STRUT* is reading the data:

file_type -s This field explain what's in the file. It defaults to sample.

7.2 Strut File Types

You will find in this section some examples of file types. This is not an exhaustive list, as new file types can be added as will.

7.3 Samples

Example from TIMIT database:

```
NIST_1A
   1024
file_type -s7 samples
database_id -s5 TIMIT
database_version -s3 1.0
utterance_id -s11 jjb0_si1277
channel_count -i 1
sample_count -i 36864
sample_rate -i 16000
sample_min -i -1634
sample_max -i 1735
sample_n_bytes -i 2
sample_byte_format -s2 01
sample_sig_bits -i 16
sample_coding -s none | alaw | ulaw | shorten
end_head
```

Note: The file_type can be omitted in this kind of files, so that the sample files coming from LDC are compatible. The sample_coding field explains how the samples must be interpreted. Usually, the lossless *shorten* format will be used.

7.4 Features

The first step of a recognition chain, right after the segmentation of the input stream by a voice activity detector, is to compute acoustic features.

```
STRUT_1A
   1024
file_type -s8 features
feature_type -s8 Cepstrum
database_id -s5 TIMIT
database_version -s3 1.0
utterance_id -s11 jjb0_si1277
features -s9 rasta-plp
frame_count -i 230
frame_rate -i 10
window_size -i 20
```

```
feature_dimension -i 26
data_format -s9 BigEndian
end_head
```

You will find also a lot of optional fields (depending on the feature extraction type) that will allow to re-compute the features with exactly the same parameters

Feature type can be one of:

Cepstrum

CMSCepstrum

LifteredCepstrum

LifteredCMSCepstrum

Plp

RastaPlp

7.5 Labels

Although discrete models are seldom used, labels files are described here, because they were the sole segmentation files in *STRUT1*.

A typical header:

```
STRUT_1A
   1024
file_type -s6 labels
database_id -s5 TIMIT
database_version -s3 1.0
utterance_id -s11 jjb0_si1277
frame_count -i 230
clustering -s3 LBG
codebook_count -i 3
codebook_format -s4 0012
data_format -s9 BigEndian
codebook0 -s3 plp
codebook0_size -i 512
codebook1 -s9 delta_plp
codebook1_size -i 128
codebook2 -s31 delta_energy+delta_delta_energy
codebook2_size -i 64
end_head
```

The labels must be stored with the minimal storing requirement possible. For codebook size up to 256, the labels can be stored on one byte. For greater codebook sizes, two bytes are necessary. The field codebook_format in the header shows how the labels are stored. For example,

0012 means that codebook 0 is stored on the two first bytes (don't forget that data_format tells the order of those two bytes), that codebook 1 is stored on the third byte, and that codebook 2 is stored on the fourth.

7.6 Segmentation

In *STRUT1*, there was no segmentation format per se. One uses the more general label format, which labels each frame with the mlp output index. This unfortunately makes the file dependent on the frame shift. A segmentation format will be defined, which will allow to store the segments boundaries in milliseconds. A block will read those segmentation files and transform them into label streams, for use in the training programs.

7.7 Probabilities

```
STRUT_1A
   1024
file_type -s13 probabilities
database_id -s5 TIMIT
database_version -s3 1.0
models_id -s6 dtimit
models_version -s3 2.1
utterance_id -s11 jjb0_si1277
frame_count -i 230
data_format -s9 BigEndian
probability -s8 discrete
probability_count -i 41
phoneme_set_id -s5 TIMIT
probability_coding -s4 none
end_head


  probability -s
```

Probability description: discrete, mlp, ...

```
probabilty_coding -s none or log or lna8.
probability_count -i    number of output probabilities
```

Chapter 8

Training Perl Scripts

As MLPs are trained via an iterative use of programs like `mlp-train`, `mlp-cross-validate`, `recognize`, a tool is provided to automatically generate perl scripts that will call those programs with appropriate parameters (fig 8.1).

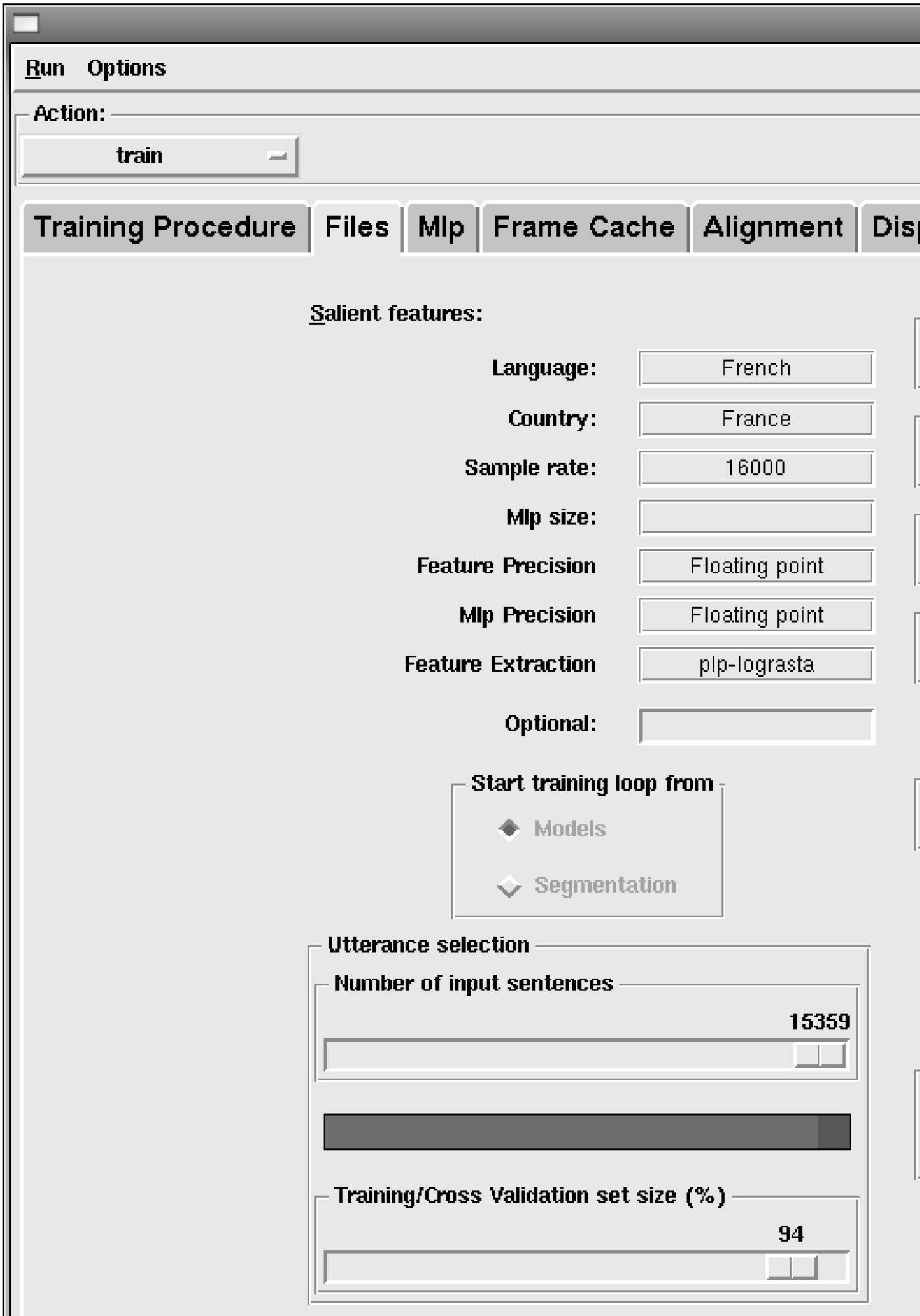
This tool will also check if the files specified are consistent. For instance, every utterance in the input sample file must appear in the segmentation file and/or the application file.

You create an initial perl script, the modify it if you need any customization for your experience, and finally run it.

If you are using zsh, you might run your script like this:

```
my-training-script.pl &!
```

The ampersand tells the shell to run the command in the background, and the exclamation mark says the terminal must disown the job. Thsi way you can close your window (and shutdown your laptop, if you run the script an a remote machine), the job will continue to run in the background.


Run Options
Action:
train
Training Procedure
Files
Mlp
Frame Cache
Alignment
Salient features:
Language:
French
Country:
France
Sample rate:
16000
Mlp size:
Feature Precision
Floating point
Mlp Precision
Floating point
Feature Extraction
plp-lograsta
Optional:
Start training loop from
Models
Segmentation
Utterance selection
Number of input sentences
15359
Training/Cross Validation set size (%)
94

Appendix A

Install STRUT

A.1 Software Installation

A.1.1 Install From Sources

A.1.2 Install From CVS

A.1.3 Install Binary Package

A.2 File System Organization

This chapter describe how the databases are organized at the *STRUT* development site. Although keeping the same hierarchy is not mandatory, it is always better to follow guidelines. You can choose the name of the database, and replace `/asr/database` by whatever you want.

The databases are installed in subdirectories of `/asr/database`, following these rules:

- the main directory name reflects the language. By convention, languages are specified by three charachters: the first two defines the language (e.g. ***fr*** for french, ***en*** for english), the third one is the first letter of the english name of a country. Examples:

  frf french spoken in France
  enu english spoken in the US
  eng english spoken in Great Brittain
  sws swedish spoken in Sweden
  tut turkish spoken in Turquey

- Then comes the database id. So , there will be directories like:
  - `/asr/database/frf/bref`
  - `/asr/database/eng/wsjcam0`
  - `/asr/database/enu/wsj`
  - `/asr/database/enu/rm1`
  - `/asr/database/enu/rm2`

- For purposes of a good organization of related data, there is a `/asr/database/mul` directory, which will contain multilingual database like *AURORA*, or home made test databases.

A.2.1 Database Subdirectory Organization

In a database sub-directory, you find those files or directories:

README an ASCII file explaining the database contents. README should be written in capital letters, so it will appear before the directory names in a sorted list. Please don't explain the directory structure if it follows the conventions. You must only describe what differs from the standard.

application/ all the files that are needed to build an application for a recognizer (and the application file themselves). The following conventions have bee adopted for the filenames extensions:

- ***.app** application files, the output of `compile-asr`.
- ***.fsg** a Finate State Grammar file. This file contains only the FSG, and will be rarely used
- ***.lxd** the same as an FSG file, with aditional information to help building the application, such as the language, and some phonetic transcriptions.
- ***.wpg** word pair grammar: words that can follow each other, in the *Y0* format.
- ***.dic** dictionaries
- ***.hmm** phoneme topologies

features/ the database after feature extraction. The training program is normally able to compute the feature on-line, so this directory should usually be empty. However it sometimes can be useful to pre-process the database, for instance when you are experimenting new features. Also, in order to compute the features, mlp-train needs a template STRUT file, from which it reads the header. This is a good place to put such files. The extension should reflect the main feature extraction parameters, like `.plp-lograsta` or `.lpc-jrasta`. Different options must be seperated by a dash:

- ***.plp** Perceptual Linear Predictive speech analysis
- ***.mel** mel frequency scale cepstral coefficients
- ***.lpc** linear predictive coefficients
- ***.*-rasta** rasta fileetring
- ***.*-lograsta** log-rasta
- ***.*-jrasta** J rasta
- ***.*-spsub** spectral subtraction
- ***.*-wiener** wiener filtering

probabilities/ (temporary) files containing state probabilities (output of MLP).

models/ the filename should reflect as much as possible what this model is good for, for instance: frf08.180-0600-036.plp-jasta.mlp.

- ***.mlp** multilayer perception weights file

***.gmm** gaussian mixture models

phonetic/ the file needed to perform an alignment. STRUT1 old 'phonetic' files should go there, but it is expected that they will slowly disappear. Here are the conventions for the STRUT2 files:

***.app** the (archive of) app file, as needed by the program `recognize` to perform a segmentation (state alignment) of a database.

***.lxd** the fsg files used to created the app's.

reference/ the references for the test databases

***.ref** orthographic description of the test sentences

result/ the speech recognition experiments outputs. You will find such files as:

***.hyp** the hypothesis (the output of `recognize`)

***.sgml** the results processed by `sclite`

***.html** an html document describing the experiments and the results

samples/ these are the samples files. As much as possible, they should be stored in the *shorten* format. Since this files seldom change and can be pretty big, this directory is never backed up. The following conventions can help to describe the format of the file:

***.sam** a *STRUT* archive, with an undefined sample coding.

***.shn** the same, but with samples coded with the *shorten* algorithm.

***.pcm** raw data in PCM format

***.wav** samples in Micro$oft wave file format. They can be read by any *STRUT* program.

script/ any script related to the database (training script, database handling scripts).

segmentation/ the segmentation of the database. They can come in two formats:

***.seg** the STRUT2 segmentation format, which is independent of the frame shift and the sampling frequency.

***.lbl** the STRUT1 format, which labels the frames. Unfortunately, this kind of file depends on the frame shift.

setup/ any setup of a STRUT program.

text/ files coming from miscellaneous sources, database cdroms or internet, unprocessed.

Appendix B

Environment Variables

Appendix C

Examples of HMM topology

The list and the topology of HMMs is given in the HMM topology file (`*.hmm`) which is typically located in the `$DATADIR/application` directory. Two examples of HMM topology files are presented in the following sections, word-based HMMs and phoneme-based HMMs, respectively. We describe here the format of these files. Every HMM topology file begins with the tag `PHONE`, directly followed by the number of HMMs in the file. Then comes an optional line which indicates the silence state identifier. This line is required if the silence state identifier is different from `0`. The remainder of the file contains the description of the topologies of the HMMs.

Figure C.1(a) shows the topology of the word-based HMM for word "`eight`". First, we have the line:

```
0 13 eight
```

This HMM has `0` as identifier and `eight` as label. It counts 13 states. Then, we found a list of 13 state identifiers:

```
-1 -2 0 0 1 1 2 2 3 3 4 4 5
```

The first two state identifiers correspond to the entry (`-1`) and exit (`-2`) states. They are dummy states (*i.e.*, not associated with any statistical distribution for emission of acoustic vectors) which serve as connecting states between HMMs. Next, we find the state identifiers of the active states. Every state identifier corresponds to a position in the probability vector given by the acoustic model, the output of the MLP. Then, we find the description of the connections between the states:

```
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
...
```

In this example, the entry state located at position `0` in the list of state identifiers allows a single transition to the state located at position `2` in the list of state identifiers. Naturally, this transition has a probability equal to one. The exit state has located at position `1` in the list of state identifiers no transition possible. The first active state located at position `2` in the list of state identifiers has only one transition to the second active state. The second active state

(a)

```
0 13 eight
-1 -2 0 0 1 1 2 2 3 3 4 4 5
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 12 0.5
12 2 12 0.5 1 0.5
```

(b)

```
22 9 ey
-1 -2 22 22 22 22 22 22 22
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 1 5 1.0
5 1 6 1.0
6 1 7 1.0
7 1 8 1.0
8 2 8 0.5 1 0.5
```

Figure C.1: Topology of a left-to-right HMM for (a) word "`eight`" and (b) phoneme "`ey`".

has two transitions, one to the third active state and as a self-transition, both are equally likely. And so on. Figure 1.2(a) depicts a graphical view of this HMM. Likewise, we can describe the topology of the phoneme-based HMM for the phoneme "`ey`" given in figure C.1(b) and depicted in figure 1.2(b).

C.0.2 A Word-Based HMM set

We give here a typical HMM topology file for word-based HMMs of English digits: `aurora2-words.hmm`,

```
PHONE
12

#SilenceState 72

0 13 eight
-1 -2 0 0 1 1 2 2 3 3 4 4 5
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 12 0.5
12 2 12 0.5 1 0.5

1 19 five
-1 -2 6 6 7 7 8 8 9 9 10 10 11 11 12 12 13 13 14
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 12 0.5
12 1 13 1.0
13 2 13 0.5 14 0.5
14 1 15 1.0
15 2 15 0.5 16 0.5
16 1 17 1.0
17 2 17 0.5 18 0.5
18 2 18 0.5 1 0.5

2 16 four
-1 -2 15 15 16 16 17 17 18 18 19 19 20 20 21 21
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 12 0.5
12 1 13 1.0
13 2 13 0.5 14 0.5
14 1 15 1.0
15 2 15 0.5 1 0.5

3 12 nine
-1 -2 22 22 23 23 24 24 25 25 26 26
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 1 0.5

4 12 oh
-1 -2 27 27 28 28 29 29 30 30 31 31
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 1 0.5

5 12 one
-1 -2 32 32 33 33 34 34 35 35 36 36
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 1 0.5

6 17 seven
-1 -2 37 37 38 38 39 39 40 40 41 41 42 42 43 43 44
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 12 0.5
12 1 13 1.0
13 2 13 0.5 14 0.5
14 1 15 1.0
15 2 15 0.5 16 0.5
16 2 16 0.5 1 0.5

7 15 six
-1 -2 45 45 46 46 47 47 48 48 49 49 50 50 51
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 12 0.5
12 1 13 1.0
13 2 13 0.5 14 0.5
14 2 14 0.5 1 0.5

8 16 three
-1 -2 52 52 53 53 54 54 55 55 56 56 57 57 58 58
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 12 0.5
12 1 13 1.0
13 2 13 0.5 14 0.5
14 1 15 1.0
15 2 15 0.5 1 0.5

9 12 two
-1 -2 59 59 60 60 61 61 62 62 63 63
0 1 2 1.0
```

```
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 1 0.5

10 17 zero
-1 -2 64 64 65 65 66 66 67 67 68 68 69 69 70 70 71
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 4 0.5
4 1 5 1.0
5 2 5 0.5 6 0.5
6 1 7 1.0
7 2 7 0.5 8 0.5
8 1 9 1.0
9 2 9 0.5 10 0.5
10 1 11 1.0
11 2 11 0.5 12 0.5
12 1 13 1.0
13 2 13 0.5 14 0.5
14 1 15 1.0
15 2 15 0.5 16 0.5
16 2 16 0.5 1 0.5

11 5 _
-1 -2 72 72 72
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5
```

C.0.3 A Phoneme-Based HMM set

We give here a typical HMM topology file for phoneme-based HMMs of English phonemes: `aurora2-phonemes.hmm`.

```
PHONE
33

#SilenceState 32

0 5 b
-1 -2 0 0 0
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

1 5 d
-1 -2 1 1 1
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

2 5 t
-1 -2 2 2 2
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

3 4 k
-1 -2 3 3
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 1 0.5

4 5 bcl
-1 -2 4 4 4
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

5 5 dcl
-1 -2 5 5 5
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

6 4 tcl
-1 -2 6 6
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 1 0.5

7 4 kcl
-1 -2 7 7
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 1 0.5

8 5 s
-1 -2 8 8 8
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

9 5 sh
-1 -2 9 9 9
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

10 5 z
-1 -2 10 10 10
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

11 4 f
-1 -2 11 11
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 1 0.5

12 4 th
-1 -2 12 12
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 1 0.5

13 4 v
-1 -2 13 13
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 1 0.5

14 5 n
-1 -2 14 14 14
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

15 5 l
-1 -2 15 15 15
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

16 5 r
-1 -2 16 16 16
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

17 7 w
-1 -2 17 17 17 17 17
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 1 5 1.0
5 1 6 1.0
6 2 6 0.5 1 0.5

18 5 hh
-1 -2 18 18 18
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

19 8 iy
-1 -2 19 19 19 19 19 19
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 1 5 1.0
5 1 6 1.0
6 1 7 1.0
7 2 7 0.5 1 0.5

20 5 ih
-1 -2 20 20 20
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

21 6 eh
-1 -2 21 21 21 21
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 1 5 1.0
5 2 5 0.5 1 0.5

22 9 ey
-1 -2 22 22 22 22 22 22 22
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 1 5 1.0
5 1 6 1.0
6 1 7 1.0
7 1 8 1.0
8 2 8 0.5 1 0.5

23 5 ae
-1 -2 23 23 23
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

24 5 aw
-1 -2 24 24 24
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

25 11 ay
-1 -2 25 25 25 25 25 25 25 25 25
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 1 5 1.0
5 1 6 1.0
6 1 7 1.0
7 1 8 1.0
8 1 9 1.0
9 1 10 1.0
10 2 10 0.5 1 0.5

26 4 ah
-1 -2 26 26
0 1 2 1.0
1 0
2 1 3 1.0
3 2 3 0.5 1 0.5

27 7 ao
-1 -2 27 27 27 27 27
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 1 5 1.0
5 1 6 1.0
6 2 6 0.5 1 0.5

28 9 ow
-1 -2 28 28 28 28 28 28 28
0 1 2 1.0
1 0
2 1 3 1.0
```

```
3 1 4 1.0
4 1 5 1.0
5 1 6 1.0
6 1 7 1.0
7 1 8 1.0
8 2 8 0.5 1 0.5

29 5 uw
-1 -2 29 29 29
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

30 5 er
-1 -2 30 30 30
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

31 5 ax
-1 -2 31 31 31
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5

32 5 _
-1 -2 32 32 32
0 1 2 1.0
1 0
2 1 3 1.0
3 1 4 1.0
4 2 4 0.5 1 0.5
```

Bibliography

[1] L. Rabiner and B.-H. Juang, "Fundamentals of Speech Recognition", *Prentice-Hall*, 1993.

[2] R. Boite, H. Bourlard, T. Dutoit, J. Hancq, H. Leich, "Traitement de la Parole", 2nd Edition, *Presses Polytechniques Universitaires Romandes*, 2000.

[3] J. W. Picone, "Signal Modeling Techniques in Speech Recognition", *Proceedings of the IEEE*, vol. 81, no. 9, pp. 1214–1247, Sep. 1993.

[4] H. Hermansky, "Perceptual Linear Predictive (PLP) Analysis of Speech", *Journal of the Acoustical Society of America*, vol. 87, no. 4, pp. 1738–1752, Apr. 1990.

[5] , H. Hermansky and N. Morgan, "RASTA Processing of Speech", *IEEE Transactions on Speech and Audio Processing*, vol. 2, no. 4, pp. 578–589, Oct. 1994.

[6] S. Furui, "Cepstral Analysis Technique for Automatic Speaker Verification", *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 29, no. 2, pp. 254–272, Apr. 1981.

[7] S. F. Boll, "Suppression of Acoustic Noise in Speech Using Spectral Subtraction", *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 27, no. 2, pp. 113–120. Apr. 1979.

[8] J. S. Lim and A. V. Oppenheim, "Enhancement and Band-width Compression of Noisy Speech", *Proceedings of the IEEE*, vol. 67, no. 12, pp. 1586–1604, Dec. 1979.

[9] H. Bourlard and N. Morgan, "Connectionist Speech Recognition, *Kluwer Academic Publishers*, 1994.

[10] Perl Language Website – `http://www.perl.org`.

[11] Perl / Tk Website – `http://www.perltk.org`.

[12] Tcl Developer Site – `http://www.tcl.tk`.

[13] Python Language Website – `http://www.python.org`.

[14] Aurora 2.0 Webpage – `http://www.elda.fr/proj/aurora2.html`.